Generalization-Enhanced Code Vulnerability Detection via Multi-Task Instruction Fine-Tuning

Xiaohu Du[1*†], **Ming Wen**[1*†‡§], **Jiahao Zhu**[1*†], **Zifan Xie**[1*†], **Bin Ji**[3], **Huijun Liu**[3], **Xuanhua Shi**[2*¶], **Hai Jin**[2*¶]

[1]School of Cyber Science and Engineering, Huazhong University of Science and Technology
[2]School of Computer Science and Technology, Huazhong University of Science and Technology
[3]College of Computer, National University of Defense Technology
{xhdu, mwenaa, m202271745, xzff, xhshi, hjin}@hust.edu.cn,
{jibin, liuhuijun}@nudt.edu.cn

Abstract

Code Pre-trained Models (CodePTMs) based vulnerability detection have achieved promising results over recent years. However, these models struggle to generalize as they typically learn superficial mapping from source code to labels instead of understanding the root causes of code vulnerabilities, resulting in poor performance in real-world scenarios beyond the training instances. To tackle this challenge, we introduce VulLLM, a novel framework that integrates multi-task learning with *Large Language Models* (LLMs) to effectively mine deep-seated vulnerability features. Specifically, we construct two auxiliary tasks beyond the vulnerability detection task. First, we utilize the vulnerability patches to construct a vulnerability localization task. Second, based on the vulnerability features extracted from patches, we leverage GPT-4 to construct a vulnerability interpretation task. VulLLM innovatively augments vulnerability classification by leveraging generative LLMs to understand complex vulnerability patterns, thus compelling the model to capture the root causes of vulnerabilities rather than overfitting to spurious features of a single task. The experiments conducted on six large datasets demonstrate that VulLLM surpasses seven state-of-the-art models in terms of effectiveness, generalization, and robustness.

1 Introduction

Code Pre-trained Models (CodePTMs) such as CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), and UniXcoder (Guo et al., 2022) have been increasingly applied to automated code vulnerability detection over recent years, achieving *state-of-the-art* (SOTA) results. In particular, these CodePTMs take code snippets as inputs and predict whether potential vulnerabilities exist in the code. However, a recent study (Du et al., 2023a) has highlighted a critical limitation in these models' generalization capabilities, particularly when dealing with *out-of-distribution* (OOD) data. The limitation arises as existing approaches tend to capture superficial rather than in-depth vulnerability features when learning the mapping from source code to labels. A notable manifestation is the inability of such approaches to accurately differentiate adversarial examples (Zhang et al., 2023b, 2020) that merely replace identifiers, indicating that their predictions are affected by factors that are irrelevant to the vulnerability. Furthermore, the learning paradigm via mapping from source code to labels struggles with the generalization ability when handling vulnerable code from multiple projects, as the code from different projects often varies in programming style and application contexts, thus leading to divergent distributions of vulnerability features (Du et al., 2023a).

It is noteworthy that recently emerged *Large Language Models* (LLMs) have demonstrated remarkable reasoning and generalization capabilities (Ge et al., 2023; Cao et al., 2023) across various domains, thus inspiring us to harness them for developing more robust vulnerability detection models. However, directly applying LLMs in code vulnerability detection encounters various challenges due to the absence of specialized training tailored to this particular task (Zhang et al., 2023a; Gao et al., 2023). Unfortunately, simply employing methods similar to CodePTMs to fine-tune LLMs will still introduce the above issues with generalization. To tackle these challenges with LLMs, this study presents VulLLM, a novel technique that integrates code vulnerability knowledge into LLMs

* National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, HUST, Wuhan, 430074, China
† Hubei Engineering Research Center on Big Data Security, Hubei Key Laboratory of Distributed System Security, HUST, Wuhan, 430074, China
‡ JinYinHu Laboratory, Wuhan, 430077, China
§ Corresponding author
¶ Cluster and Grid Computing Lab, HUST, Wuhan, 430074, China

Findings of the Association for Computational Linguistics: ACL 2024, pages 10507–10521

through instruction tuning (Zhang et al., 2023d).

To prevent the model from learning spurious features, we employ the multi-task learning paradigm to enable LLMs to learn deep-seated features rather than spurious ones. The insight is to enhance the mapping from source code to labels by adding two auxiliary tasks which aim to gain a deeper understanding of vulnerabilities by identifying their root causes and locating the corresponding vulnerable code elements. The first auxiliary task, vulnerability localization, identifies vulnerable code elements (e.g., statement) that are extracted from the patch. The second task, vulnerability interpretation, identifies vulnerabilities' root causes and outputs their textual interpretation. As no ready-made interpretation exists, we generate them using GPT-4. However, LLMs still face challenges in vulnerability detection, let alone identifying the root causes of vulnerabilities. In a manual assessment, ChatGPT barely understands half of the vulnerabilities it 'detects' based on simple prompts (Zhang et al., 2023a). To tackle this challenge, we introduce the patch-enhanced *Chain-of-Thought (Wei et al., 2022) with Self-Verification* (CoT-SV), which demonstrates effective performance to avoid error accumulation and illusions in CoT, thus enhancing the reliability of LLMs (Ni et al., 2023; Gou et al., 2023). The validations in CoT-SV include vulnerability labels, *Common Vulnerabilities and Exposures* (CVE) descriptions, and vulnerability lines with contexts extracted from the patch based on *Program Dependency Graph* (PDG) (Li et al., 2022). Auxiliary tasks enhance the variety and depth of features (i.e., the vulnerable location and root cause), thereby improving the model's comprehension of the domain knowledge of code vulnerabilities. Moreover, the diversity of features contributes variably across various tasks, compelling the model to seek solutions that perform well across all tasks, thereby preventing overfitting to specific spurious features of a single task.

In addition to the above data generated for multi-task learning, our training data also includes two real-world vulnerability datasets with the highest label accuracy from two manual evaluations (Chen et al., 2023; Croft et al., 2023): Devign (Zhou et al., 2019) and DiverseVul (Chen et al., 2023), to further enhance LLMs' learning of various code vulnerabilities. Furthermore, to verify the extensive applicability of our framework, we select three widely-used foundational models to construct VulLLM, including a general LLM, Llama-2 (Touvron et al., 2023), alongside two CodeLLMs, namely CodeLlama (Rozière et al., 2023) and StarCoder (Li et al., 2023). The results indicate that VulLLM outperforms seven existing SOTA vulnerability detection models. Overall, compared to the best baseline, UniXcoder, VulLLM demonstrates superior effectiveness with an improvement in F1 score by 8% across six datasets. Notably, within these datasets, the F1 score of VulLLM has increased by 8.58% on four OOD datasets, indicating its better generalization. Furthermore, we introduce three adversarial attacks to verify the robustness of these models. Under these attacks, the overall F1 score of VulLLM improves by 68.08% compared to UniXcoder, highlighting its enhanced robustness.

Our contributions are summarized as follows:

- **Idea.** We propose a novel perspective to leverage the interpretability of GPT-4 for generating vulnerability interpretation to enhance the vulnerability understanding of LLMs.
- **Approach.** We propose VulLLM, a framework to detect vulnerabilities with LLMs through multi-task instruction-tuning. To our best knowledge, it is the first attempt to use instruction-tuned LLMs for vulnerability detection.
- **Evaluation.** We conduct extensive experiments across six datasets and find that our approach significantly improves the effectiveness, generalization and robustness in detecting vulnerabilities. We also release the code and data at: `https://github.com/CGCL-codes/VulLLM`.

2 Related Work

2.1 Code Vulnerability Detection

Code vulnerability detection is of significant importance for the secure and stable operation of software systems. Deep learning methods can automatically learn and generalize features of vulnerabilities from extensive code samples, enabling automated inference of vulnerability patterns. This paradigm has gained widespread attention over recent years. Early vulnerability detection methods utilize *Graph Neural Networks* (GNNs) to learn vulnerability features. With the development of the Transformer (Vaswani et al., 2017) architecture, many CodePTMs, such as CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), and UniXcoder (Guo et al., 2022), have achieved better performance in vulnerability detection. They are mainly pre-trained on extensive datasets with

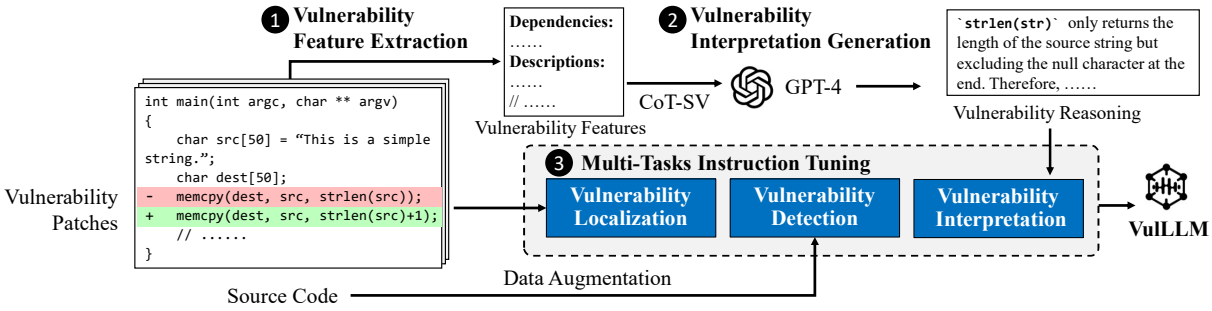


Figure 1: The general workflow of VulLLM

code and text, and have demonstrated outstanding performance in multiple code-related downstream tasks. Additionally, some approaches are built upon CodePTMs. ReGVD (Nguyen et al., 2022) encodes source code as a graph with nodes representing code tokens and features initialized based on CodePTMs. EPVD (Zhang et al., 2023c) divides the code into various execution paths based on *Control Flow Graph* (CFG) and learns different path representations based on CodePTMs. These approaches mainly train models via learning from a single-task. In this work, we utilize the paradigm of multi-task learning to enable LLMs to better understand code vulnerabilities.

2.2 Self-Verification in LLM

CoT (Wei et al., 2022) is a prompting technique to solve problems with LLMs, which employs a series of reasoning steps to tackle complex issues, akin to the thought process of humans in solving problems. Self-verification (Pan et al., 2023) aims to mitigate hallucinations (Lin et al., 2022) and unfaithful reasoning in LLMs (Golovneva et al., 2023; Lyu et al., 2023), while reducing error accumulation in CoT as well (Weng et al., 2023). Specifically, it corrects the adverse behaviors of LLMs through feedback, which also aligns with human learning strategies, that is, a cycle of attempting, making mistakes, and correcting. Verification often comes from two sources: manual validation and automatic validation. Manual validation tends to be more congruent with human preferences. For instance, InstructGPT (Ouyang et al., 2022) improves GPT-3 (Brown et al., 2020) through human feedback. Automatic validation can originate from the LLM itself or external knowledge. For instance, SelfCheck (Miao et al., 2023) demonstrates LLM's ability for correcting errors in CoT independently, without external resources. The results from different stages are employed to derive an overall confidence score, which is subsequently utilized as a weight to cast votes among multiple solutions to the same problem, thereby enhancing the accuracy of the response. Validations based on external knowledge can originate from various sources, such as Wikipedia (Varshney et al., 2023) and search engines (Gou et al., 2023). In this study, we obtain validation for vulnerability interpretation from the features extracted from the corresponding vulnerability patches (see Section 3.2 for more details).

3 Methodology

3.1 Overview

Figure 1 illustrates an overview of VulLLM, which comprises three main components: vulnerability features extraction (Section 3.2), vulnerability interpretation generation (Section 3.3), and multi-task instruction fine-tuning (Section 3.4).

3.2 Vulnerability Features Extraction

Vulnerability features serve as the essential cues for vulnerability interpretation. In this study, we aim to enhance the generalization of LLM in the context of vulnerabilities. Specifically, we explore the use of *vulnerability lines*, *vulnerability context*, and *CVE descriptions* as potential cues for understanding vulnerabilities.

Vulnerability Lines. Vulnerability lines directly point out the vulnerable code elements. We extract vulnerability lines from the patches of the vulnerable code. Patches are generated to fix existing vulnerabilities via adding or deleting certain code elements. Following existing study (Nguyen et al., 2016), we consider the deleted lines in patches to directly reflect the vulnerable semantics. For instance, if the deleted lines involve unsafe coding practices, such as *improper memory management* or *insecure input validation*, these lines could be the direct root cause of the vulnerability.

Vulnerability Context. Vulnerability context refers to the surrounding code (e.g., conditions,

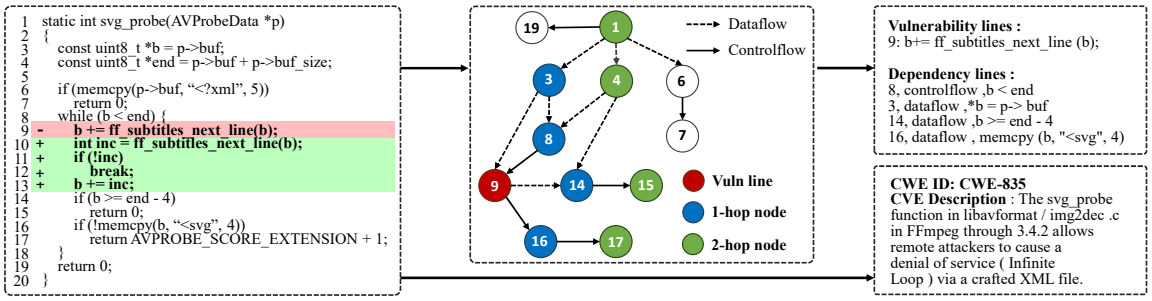


Figure 2: An example of vulnerability feature extraction

checkers, *etc.*) that provide a broader understanding of a security vulnerability. Typically, we extract code statements that have direct or indirect data dependencies and control dependencies along with the vulnerable lines as vulnerability context. To extract these code statements, we first use JOERN (JOERN, 2023) to generate the Program Dependency Graph (i.e., PDG) (Li et al., 2022) for the vulnerability functions. PDG is a directed acyclic graph where nodes represent code elements, and various types of directed edges between nodes represent relationships between code elements (e.g., if there exists a data dependency edge originating from node A and directed towards node B, it indicates that node B depends on a data variable defined at node A). PDG has been widely utilized in the domain of vulnerability detection (Li et al., 2021; Zhang et al., 2023c). Specifically, we start from the nodes corresponding to the vulnerabilities and identify neighboring nodes within a k-hop distance through both data dependency edges and control dependency edges (either outgoing or incoming). The code lines corresponding to these nodes are then added to the vulnerability context. Figure 2 illustrates an example of vulnerability CVE-2018-7751, which belongs to the type of CWE-835. The left side depicts the applied patch, the middle section showcases the corresponding PDG, and the right side displays the extracted vulnerability features. For the vulnerability line at line 9, there is a control dependency edge from line 8 to line 9, as well as three data dependency edges—one from line 3 to line 9, one from line 9 to line 14 and another from line 9 to line 16. If k is set to 1, the contextual scope encompasses statements found at lines 8, 3, 14, and 16. The parameter k is used to control the length of the generated vulnerability context, as dependency relationships in real-world code can be particularly complex. In our implementation, the value of k is set to 1, considering the limited input length capacity of LLMs.

CVE Descriptions. This information shed lights on the root causes of vulnerabilities, as they comprehensively detail common security weaknesses in software and hardware. These descriptions are invaluable for understanding the root causes of vulnerabilities, which often offer relevant background and context, explaining how these weaknesses come about and how they might be exploited under different circumstances. To collect the CVE descriptions, we have scraped them for each CVE from the NVD (NVD, 2023).

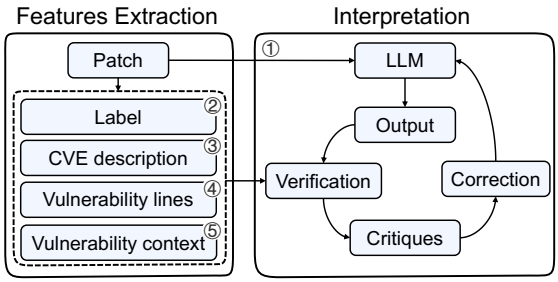


Figure 3: The implementation of CoT with Self-Verification. Numbered circles denotes the five steps

3.3 Vulnerability Interpretation Generation

The vulnerability features extracted in the previous section serve as the critical information for validating the output of each step in CoT. Figure 3 illustrates the implementation of CoT-SV.

Step 1. Given the demonstrated efficacy of role-playing in prompt engineering (Kong et al., 2023; Shanahan et al., 2023), our initial step involves adopting a role-centric prompting strategy, specifically focusing on vulnerability detection, to ensure that the model remains concentrated on the task throughout the workflow. We use the following prompt as adapted from a prior work (Zhang et al., 2023a) in this step.

> **Prompt**$_1$: `I want you to act as a vulnerability detection model. Is the following program buggy? [Code]`

where [Code] refers to the potentially buggy code. Previous study indicates that the accuracy of LLM under such prompt is suboptimal (Cheshkov et al.,

2023). Fortunately, we have the ground truth for each code. Therefore, if the LLM produces an incorrect output, we can address it in subsequent steps. More importantly, this initial step serves to reinforce LLM's acknowledgment of the presence of vulnerabilities, facilitating subsequent vulnerability reasoning. To prevent overfitting, we execute **Step 1** for an equal number of vulnerable and non-vulnerable code examples to generate vulnerability explanations. For non-vulnerable code, GPT-4 will provide interpretations indicating the absence of vulnerabilities, which will be used to construct the dataset for non-vulnerable code in the vulnerability explanation task. For vulnerable code, more precise vulnerability interpretations will be obtained through continuous verification of GPT-4's outputs in subsequent steps.

Step 2 - Step 4. We adapt a unified prompt template for various vulnerability features based on existing Self-Verification templates (Pan et al., 2023; Ling et al., 2023). The validity of the output from each individual step is verified using a directive composed of the following components: (1) information required to be verified for the current step. (2) an instruction for validity verification, such as *Please double-check the answer and analyze its correctness.* (Ling et al., 2023) (3) requirements for the output of the subsequent step under the LLM. Based on the above design, the prompts for obtaining vulnerability interpretations are as follows:

Prompt$_2$: `This program is buggy. Please double-check the answer and analyze its correctness. Next, please give the description of the vulnerability.`
Prompt$_3$: `The description of vulnerability is` **`[CVE description]`**`. Please double-check the answer and analyze its correctness. Next, please provide the lines of code that are directly pertinent to the identified vulnerability.`
Prompt$_4$: `The vulnerability lines are` **`[Vulnerability lines]`**`. Please double-check the answer and analyze its correctness. Next, please provide the data dependency and control dependency lines related to the vulnerability lines.`

where [CVE description] and [Vulnerability lines] denote the vulnerability features as extracted in the previous subsection.

Step 5. Vulnerability context constitutes the final features for verification within CoT-SV. Following this verification, we employ the LLM to synthesize the vulnerability interpretation by integrating the aforementioned categories of features. Specifically, we instruct the LLM to generate a vulnerability interpretation that refers to the vulnerability lines and vulnerability context.

Prompt$_5$: `The dependency lines are` **`[Dependency lines]`**`. Please double-check the answer and analyze its correctness. Next, considering the vulnerability's description, please present the vulnerability interpretation by referring to the vulnerable and dependent lines.`

The results generated by CoT-SV encompass rich knowledge specific to vulnerabilities. For instance, the CVE description constrains the high-level overview of the vulnerability, whereas the vulnerability lines pinpoint its precise location. Furthermore, the dependency lines characterize the context of vulnerabilities. Such knowledge ensures that the reasoning process integrates an extensive repository of domain-specific knowledge on vulnerabilities. Concurrently, CoT-SV further abstracts the data flow and control flow of vulnerabilities, translating the dependency lines into natural language descriptions of the vulnerability context. Finally, all the generated data are manually verified to exclude instances where the final judgment of GPT-4 is still incorrect. We take the vulnerable code in Figure 2 as an example and present different interpretations of **Step 1** and the entire CoT-SV generation process separately in Appendix A.

3.4 Multi-Task Instruction Fine-tuning

Data Preparation. The data serving for the above multi-task data originates from the PatchDB (Wang et al., 2021), which contains the patch information, a feature not commonly found in other datasets. The diffs in patches are directly associated with vulnerabilities and essential for obtaining vulnerability interpretations. However, this patch-based vulnerability dataset is limited in its quantity and insufficient for LLMs to learn a broad range of vulnerability patterns. To infuse LLMs with more vulnerability knowledge, we further incorporate two additional datasets for the vulnerability detection task: DiverseVul (Chen et al., 2023) and Devign (Zhou et al., 2019), which are the two real-world datasets with the highest label accuracy in manual evaluation sampling (Croft et al., 2023; Chen et al., 2023). Increasing the data volume for the primary task ensures that the model does not deviate from the core objective while learning auxiliary tasks, which also helps in maintaining the priority of the primary task. These two datasets contain 797 distinct projects. A common challenge when training on multi-source

code datasets is how to handle the variation of feature distribution among different projects. To mitigate this variation, we employ random identifier substitution to enhance the generalization of LLMs on multi-project code. The core idea is to reduce model dependence on a specific project's features by increasing data diversity, thereby minimizing the risk of overfitting and enhancing the model's adaptability to different coding styles. Specifically, we replace 10% of the existing identifiers within the original code with randomly chosen identifiers sourced from the complete dataset. The input for all three tasks is the source code. For the output, vulnerability detection yields a label of 0 or 1. Vulnerability localization identifies the vulnerable line as extracted in Figure 2, and vulnerability interpretation provides the natural language (the final result of CoT-SV, as demonstrated in Appendix A).

Instruction Fine-tuning. Instruction fine-tuning aims to optimize the response of LLMs to specific directives, thus ensuring the alignment with the requirements of a particular task. Specifically, we employ instruction fine-tuning to train a more specialized, adaptable, and efficient LLM for vulnerability detection. For each task, we provide a distinct instruction. By integrating this instruction with the input code, the LLM is capable of producing specific outputs. Subsequently, the LLM quantifies the discrepancy between the generated output and the anticipated target, leveraging this deviation to fine-tune the weights of LLM. In this work, we adapt the template provided by Alpaca (Taori et al., 2023) for instruction fine-tuning:

```
Below is an instruction that describes a task,
paired with an input that provides further
context. Write a response that appropriately
completes the request.
### Instruction:
[Task Prompt]
### Input:
[Input]
### Response:
[Output]
```

where [Input] and [Output] are obtained from the above **Data Preparation**, and the [Task Prompt] directs LLMs to generate task-specific outputs based on different tasks. We provide specific examples of instruction data for different tasks in Appendix A.

4 Experimental Setup

4.1 Datasets

We select six widely-used C/C++ vulnerability detection datasets to evaluate different models extensitively: DiverseVul (Chen et al., 2023), Devign (Zhou et al., 2019), BigVul (Fan et al., 2020), CVEfixes (Bhandari et al., 2021), ReVeal (Chakraborty et al., 2022), and Juliet (Boland and Black, 2012). The first two datasets are involved in model training (denoted as $Dataset_1$) while the latter four datasets are not presented in the training process (denoted as $Dataset_2$). Therefore, the results on $Dataset_2$ can further reflect the models' generalizability besides effectiveness. The details of the datasets can be found in Appendix B.

4.2 Baselines

In our evaluation, we compare VulLLM with the following SOTA models, encompassing diverse architectures and approaches to ensure a broad spectrum of comparison. Specifically, our baselines include two GNNs-based models: Devign (Zhou et al., 2019) and ReVeal (Chakraborty et al., 2022), three CodePTMs: CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), UniXcoder (Guo et al., 2022), and two models based on CodePTMs: ReGVD (Nguyen et al., 2022) and EPVD (Zhang et al., 2023c) that are specifically designed for vulnerability detection. The details of these baselines can be found in Appendix C. The implementation details of all the baselines and our approach can be found in Appendix E.

5 Results

5.1 Effectiveness and Generalization

We select two versions with different parameter sizes of Llama-2, CodeLlama, and StarCoder as our base models respectively, to validate the extensive applicability of our framework. Specifically, we evaluate the effectiveness of various approaches across the six test datasets. Notably, $Dataset_2$, which is not involved in model training, can further reflect the model's generalization.

The results shown in Table 1 indicate that VulLLM, based on CodeLlama-13B, exhibits the highest performance, yielding an overall F1 score of 66.54%. Across all models and datasets, VulLLM based on CodeLlama-13B consistently ranks either first or second, which outperforms the 7 selected baselines. Therefore, our subsequent analysis of VulLLM is based on CodeLlama-13B. Compared to the best baseline model, UniXcoder, VulLLM demonstrates an overall effectiveness improvement by 8% (i.e., (66.54-61.61)/61.61) across all six

Methods	Size	$Dataset_1$		$Dataset_2$				Average		
		DiverseVul	Devign	BigVul	CVEfixes	ReVeal	Juliet	$Dataset_1$	$Dataset_2$	All
Devign	1M	60.36	58.93	53.14	55.09	59.43	57.71	59.65	56.34	57.44
ReVeal	1M	57.04	53.84	53.92	53.49	56.67	57.86	55.44	55.49	55.47
CodeBERT	125M	65.45	66.81	55.19	56.03	46.48	5.36	66.13	40.77	49.22
GraphCodeBERT	125M	66.00	66.62	54.70	58.82	55.61	31.22	66.31	50.09	55.50
UniXcoder	126M	67.27	66.05	56.06	59.35	59.57	61.36	66.66	59.09	61.61
ReGVD	125M	65.86	61.14	58.55	60.86	54.94	36.87	63.50	52.81	56.37
EPVD	125M	66.85	66.76	52.89	57.73	48.95	25.86	66.81	46.36	53.17
VulLLM-L2	7B	65.31	66.24	57.84	55.47	52.80	64.08	65.78	57.55	60.29
	13B	70.42	70.29	59.54	61.48	57.79	57.04	70.36	58.96	62.76
VulLLM-SC	7B	60.43	63.46	62.31	**64.50**	50.81	63.32	61.95	60.24	60.81
	15B	62.02	62.77	46.17	52.09	59.48	**69.50**	62.40	56.81	58.67
VulLLM-CL	7B	68.23	67.84	63.51	59.26	**66.45**	58.84	68.04	62.02	64.02
	13B	**70.99**	**71.63**	**64.42**	61.92	64.52	65.77	**71.31**	**64.16**	**66.54**

Table 1: The F1 scores on six datasets. The abbreviations "L2", "SC", and "CL" refer to the Llama-2, StarCoder, and CodeLlama, respectively. The best results are highlighted in bold, while the next best results are underlined

datasets. Notably, it exhibits improvements by 8.58% (i.e., (64.16-59.09)/59.09) in the F1 score on $Dataset_2$, indicating its superior generalization. In addition, existing models generally exhibit poor generalization ability. In the 20 (5 models $\times$ 4 datasets) generalization experiments on CodePTMs, the average F1 score of the baseline models decrease by 11.36% to 38.36% (24.33% on average) compared to $Dataset_1$. In contrast, VulLLM demonstrates a much smaller decrease in performance, decreasing by only 10.03%, and crucially, maintains F1 scores above 60% across all datasets. While GNN-based models seem to exhibit a lesser performance decline on $Dataset_2$ compared to CodePTMs, seemingly demonstrating better generalization, their poorer effectiveness and complex data preprocessing make them significantly less versatile than other models, limiting their practical applicability.

5.2 Robustness

Attack	Model	DiverseVul	ReVeal	Total Avg
MHM	UniXcoder	24.97	20.48	22.73
	VulLLM	**33.22**	**40.06**	**36.64**
WIR	UniXcoder	4.42	2.18	3.30
	VulLLM	**16.91**	**25.25**	**21.08**
DCI	UniXcoder	37.78	31.94	34.86
	VulLLM	**42.32**	**46.94**	**44.63**

Table 2: The F1 scores under adversarial attacks. For simplicity, DCI denotes the Dead Code Insertion attack

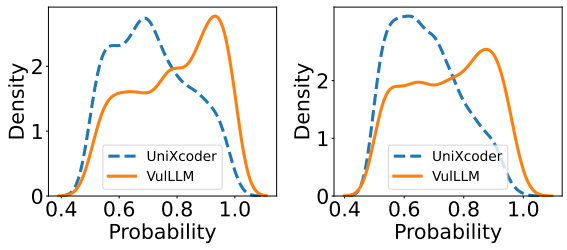
We select DiverseVul and ReVeal, from $Dataset_1$ and $Dataset_2$ respectively, in this experiment. DiverseVul is chosen due to its inclusion of a wide variety of projects, thus offering a comprehensive evaluation of the model's robustness. ReVeal is selected because its data is sourced from the Chromium and Debian packages, which differs from other datasets that originate from NVD (NVD, 2023) or GitHub repositories. We select two adversarial attacks for CodePTMs that are based on random identifier replacement: MHM (Zhang et al., 2020) and WIR-Random (Zeng et al., 2022), since they achieve the highest attack success rate in recent evaluations (Du et al., 2023b). In addition, we also construct an attack based on random dead code insertion, with the form of the dead code derived from DIP (Na et al., 2023). Since the code of DIP is not publicly available, and our objective is to compare the robustness of models under different attacks rather than pursuing the highest attack success rate, we do not directly use DIP. The details of these attacks can be found in Appendix F.

We select UniXcoder, which performs the best over all the baselines in Table 1, and VulLLM for robustness evaluation. Table 2 reveals the F1 score of two models under three adversarial attacks. It shows that UniXcoder exhibit a significant decline in performance under various adversarial attacks. Notably, the performance under ReVeal is lower than that under DiverseVul, indicating that UniXcoder has poorer robustness on OOD data. This reduced robustness is a testament to their insufficient generalization. Conversely, VulLLM demonstrates superior robustness across all attacks. Compared to UniXcoder, VulLLM demonstrates an average improvement by 68.08%. More importantly, VulLLM does not show a further decline in robustness on OOD samples, indicating that its robustness and generalization are superior to UniXcoder.

To explore the reasons behind the differences in robustness between the two models, we further examine the probability densities of correct pre-

Methods	DiverseVul	Devign	BigVul	CVEfixes	ReVeal	Juliet	Total Avg
VulLLM	70.99	71.63	64.42	61.92	64.52	65.77	66.54
w/o DA	68.89↓	65.08↓	59.80↓	61.46↓	65.65↑	66.48↑	64.56↓
w/o MT	66.62↓	66.56↓	53.42↓	59.61↓	52.87↓	63.00↓	60.35↓
w/o DA&MT	70.08↓	64.06↓	56.22↓	59.47↓	60.31↓	70.88↑	63.50↓

Table 3: Results of ablation study. The abbreviations "DA" and "MT" refer to the data augmentation and multi-task learning, respectively. ↓(↑) indicates that the performance relative to the complete VulLLM decreases (increases)



(a) DiverseVul (b) ReVeal

Figure 4: Probability density of the DiverseVul and ReVeal from VulLLM and UniXcoder

dictions made by VulLLM and UniXcoder on the two datasets. We particularly emphasize the importance of high prediction probabilities, as accurate probability predictions are especially crucial when performing safety-critical tasks. As illustrated in Figure 4, the overall performance analysis indicates that VulLLM exhibits superior probability density over UniXcoder across two datasets. On the DiverseVul dataset, VulLLM shows a significantly higher density in the high probability regions (close to 1.0), indicating stronger confidence in its results. Its curve peaks around a probability value of approximately 0.9, and then rapidly declines, suggesting a higher concentration in high probability predictions. On the ReVeal dataset, the difference in density curves between the two models is not as pronounced as in DiverseVul, but VulLLM still maintains a higher density in regions where the probability is greater than 0.8. Particularly in the probability range of 0.8 to 1.0, its density curve is above UniXcoder, peaking near a probability of 0.9. In summary, VulLLM exhibits greater confidence in its predictions, especially when providing high probability forecasts, contributing to its higher robustness.

5.3 Ablation Study

In this section, we investigate the impact of multi-task learning and data augmentation. As demonstrated in Table 3, after removing multi-task learning ("w/o MT"), the model exhibits a performance decline across all datasets, with an overall relative reduction by 9.30%. This observation underscores the pivotal role of multi-task learning within our approach, evidencing its substantial contribution towards enhancing the model's effectiveness and generalization. When the data augmentation component is removed ("w/o DA"), the model exhibits a decrease in average performance by 2.98%. Across different datasets, the model shows a decline in performance on four datasets, but an increase on ReVeal and Juliet. Such variations suggest that while the data augmentation incorporated into VulLLM effectively enhances model effectiveness, it may simultaneously impair the model's generalization on certain datasets. Finally, when two components are removed (w/o "DA&MT"), its performance decreases across five datasets. Notably, on the Devign dataset, its performance is even inferior to that of the three CodePTMs. Additionally, on both BigVul and CVEfixes, it falls short of the performance achieved by ReGVD. In summary, the ablation study clearly demonstrates the indispensable roles that multi-task learning and data augmentation play in enhancing the model's overall performance. Notably, multi-task learning emerges as the more impactful of the two components, playing a pivotal role in enhancing the model's performance.

5.4 Sensitivity to hyper-parameter

Length	Aux Data	DiverseVul	Reveal	Total Avg
512	694	70.99	64.52	66.54
1,024	1,509	69.66	64.36	66.76
2,048	2,138	68.68	67.00	66.89

Table 4: The F1 score of VulLLM under varying numbers of auxiliary task samples

Auxiliary task samples. Considering resource constraints and training efficiency, previous models are trained within a context length of 512. Consequently, the amount of auxiliary task data included is limited. To further explore the impact of the samples of auxiliary tasks on the performance of VulLLM, we expand the training context lengths to 1,024 and 2,048 to include more auxiliary task samples. We present results for two representative datasets, similar to Section 5.2, and list the aver-

age values for six datasets, as shown in Table 4. We find that an increased number of auxiliary task samples generally leads to a slight improvement in model performance, especially on OOD samples. However, there is a noticeable decline in the model's performance on in-distribution samples. The changes can be attributed to multi-task learning, where a model learns various tasks together, focusing on features common to all tasks. With more auxiliary task samples, the model adapts to diverse data, improving its overall applicability. However, this broad focus might lead to less optimal performance on specific tasks, as the model might miss finer, unique features of the original training set.

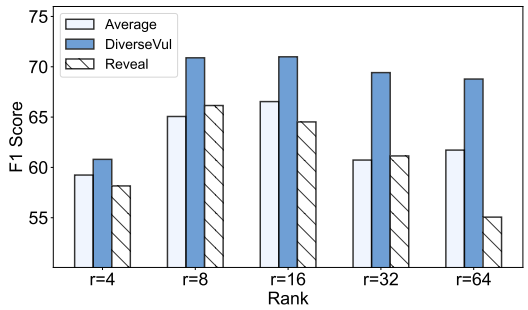


Figure 5: Average F1 score for six datasets and F1 scores for DiverseVul and Reveal at different ranks

Training parameters. To demonstrate the sensitivity of VulLLM to training parameters, we conduct experiments with different ranks in LoRA, which are proportional to the training parameters. As shown in Figure 5, we observe that the average F1 score increases as the rank increases, reaching its peak at 16. Further increasing the rank value leads to a decrease in performance, following a trend similar to existing works (Hu et al., 2022). This phenomenon may be attributed to a limited number of training parameters constraining the model's learning capacity, while an excessive number of parameters may lead to overfitting or excessive complexity in handling the parameters.

6 Conclusion

In this paper, we introduce VulLLM, a novel framework for code vulnerability detection utilizing LLMs. By innovatively integrating a vulnerability interpretation task into our multi-task learning framework alongside data augmentation strategies, we significantly enhance the LLM's capability to detect code vulnerabilities. This combination not only improves detection accuracy but also enriches the model's understanding of the context and rationale behind vulnerabilities. Extensive evaluations conducted on six diverse and comprehensive datasets demonstrate that VulLLM surpasses existing approaches in terms of effectiveness, generalization, and robustness. Further validation through ablation study confirms the critical role of multi-task learning and data augmentation in boosting VulLLM's performance.

Limitations

Due to resource limitations, our experiments are conducted on LLMs with size of 7B, 13B, and 15B, utilizing the parameter-efficient fine-tuning approach LoRA. This approach may affect the final performance. Additionally, the acquisition of vulnerability interpretations is contingent upon the capabilities inherent in the LLMs. To mitigate this limitation, we employ the SOTA LLM, GPT-4, in conjunction with CoT-SV for generating interpretations. However, there remains a potential for bias in these vulnerability explanations, particularly when dealing with code that involve complex vulnerability contexts.

Acknowledgements

We sincerely thank all anonymous reviewers for their valuable comments. This work was supported by the Major Program (JD) of Hubei Province (No.2023BAA024), the National Natural Science Foundation of China (Grant No. 62372193), and the Young Elite Scientists Sponsorship Program by CAST (Grant No. 2021QNRC001).

References

Guru Prasad Bhandari, Amara Naseer, and Leon Moonen. 2021. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2021, Athens Greece, August 19-20, 2021*, pages 30–39. ACM.

Tim Boland and Paul E. Black. 2012. Juliet 1.1 C/C++ and java test suite. *Computer*, 45(10):88–90.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess,

Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Shulin Cao, Jiajie Zhang, Jiaxin Shi, Xin Lv, Zijun Yao, Qi Tian, Juanzi Li, and Lei Hou. 2023. Probabilistic tree-of-thought reasoning for answering knowledge-intensive complex questions. *CoRR*, abs/2311.13982.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep learning based vulnerability detection: Are we there yet? *IEEE Trans. Software Eng.*, 48(9):3280–3296.

Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David A. Wagner. 2023. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2023, Hong Kong, China, October 16-18, 2023*, pages 654–668. ACM.

Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. 2023. Evaluation of chatgpt model for vulnerability detection. *CoRR*, abs/2304.07232.

Roland Croft, Muhammad Ali Babar, and M. Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 121–133. IEEE.

Qianjin Du, Shiji Zhou, Xiaohui Kuang, Gang Zhao, and Jidong Zhai. 2023a. Joint geometrical and statistical domain adaptation for cross-domain code vulnerability detection. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 12791–12800. Association for Computational Linguistics.

Xiaohu Du, Ming Wen, Zichao Wei, Shangwen Wang, and Hai Jin. 2023b. An extensive study on adversarial attack against pre-trained models of code. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 489–501. ACM.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories, MSR 2020, Seoul, Republic of Korea, 29-30 June, 2020*, pages 508–512. ACM.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Zeyu Gao, Hao Wang, Yuchen Zhou, Wenyu Zhu, and Chao Zhang. 2023. How far have we gone in vulnerability detection using large language models. *CoRR*, abs/2311.12420.

Yingqiang Ge, Wenyue Hua, Jianchao Ji, Juntao Tan, Shuyuan Xu, and Yongfeng Zhang. 2023. Openagi: When LLM meets domain experts. *CoRR*, abs/2304.04370.

Olga Golovneva, Moya Chen, Spencer Poff, Martin Corredor, Luke Zettlemoyer, Maryam Fazel-Zarandi, and Asli Celikyilmaz. 2023. ROSCOE: A suite of metrics for scoring step-by-step reasoning. In *Proceedings of the Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. CRITIC: large language models can self-correct with tool-interactive critiquing. *CoRR*, abs/2305.11738.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 7212–7225. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *Proceedings of the 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. In *Proceedings of the Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

JOERN. 2023. `https://github.com/joernio/joern`. Accessed: 2023-12.

Aobo Kong, Shiwan Zhao, Hao Chen, Qicheng Li, Yong Qin, Ruiqi Sun, and Xin Zhou. 2023. Better zero-shot reasoning with role-play prompting. *CoRR*, abs/2308.07702.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *CoRR*, abs/2305.06161.

Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, Athens, Greece, August 23-28, 2021*, pages 292–303. ACM.

Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Trans. Dependable Secur. Comput.*, 19(4):2244–2258.

Stephanie Lin, Jacob Hilton, and Owain Evans. 2022. Truthfulqa: Measuring how models mimic human falsehoods. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 3214–3252. Association for Computational Linguistics.

Zhan Ling, Yunhao Fang, Xuanlin Li, Zhiao Huang, Mingu Lee, Roland Memisevic, and Hao Su. 2023. Deductive verification of chain-of-thought reasoning. In *Proceedings of the Thirty-seventh Conference on Neural Information Processing Systems*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. *CoRR*, abs/2301.13379.

Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. 1953. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092.

Ning Miao, Yee Whye Teh, and Tom Rainforth. 2023. Selfcheck: Using llms to zero-shot check their own step-by-step reasoning. *CoRR*, abs/2308.00436.

CheolWon Na, YunSeok Choi, and Jee-Hyong Lee. 2023. DIP: dead code insertion based black-box attack for programming language model. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 7777–7791. Association for Computational Linguistics.

Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. 2022. Regvd: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*, pages 178–182. ACM/IEEE.

Viet Hung Nguyen, Stanislav Dashevskyi, and Fabio Massacci. 2016. An automatic method for assessing the versions affected by a vulnerability. *Empir. Softw. Eng.*, 21(6):2268–2297.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. LEVER: learning to verify language-to-code generation with execution. In *Proceedings of the International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 26106–26128. PMLR.

NVD. 2023. `https://nvd.nist.gov/`. Accessed: 2023-12.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Proceedings of the Advances in Neural Information Processing Systems, NeurIPS 2022*, volume 35, pages 1–13. Curran Associates, Inc.

Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. 2023. Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies. *CoRR*, abs/2308.03188.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Murray Shanahan, Kyle McDonell, and Laria Reynolds. 2023. Role-play with large language models. *CoRR*, abs/2305.16367.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Alpaca: A strong, replicable instruction-following model. *Stanford Center for Research on Foundation Models*, 3(6):7.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

Neeraj Varshney, Wenlin Yao, Hongming Zhang, Jianshu Chen, and Dong Yu. 2023. A stitch in time saves nine: Detecting and mitigating hallucinations of llms by validating low-confidence generation. *CoRR*, abs/2307.03987.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, NeurIPS 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. 2021. Patchdb: A large-scale security patch dataset. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*, pages 149–160. IEEE.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Yixuan Weng, Minjun Zhu, Fei Xia, Bin Li, Shizhu He, Kang Liu, and Jun Zhao. 2023. Large language models are better reasoners with self-verification. *CoRR*, abs/2212.09561.

Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022, Virtual Event, South Korea, July 18 - 22, 2022*, pages 39–51. ACM.

Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2023a. Prompt-enhanced software vulnerability detection using chatgpt. *CoRR*, abs/2308.12697.

Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 1169–1176. AAAI Press.

Jie Zhang, Wei Ma, Qiang Hu, Shangqing Liu, Xiaofei Xie, Yves Le Traon, and Yang Liu. 2023b. A black-box attack on code models via representation nearest neighbor search. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 9706–9716. Association for Computational Linguistics.

Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. 2023c. Vulnerability detection by learning from syntax-based execution paths of code. *IEEE Trans. Software Eng.*, 49(8):4196–4212.

Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, and Guoyin Wang. 2023d. Instruction tuning for large language models: A survey. *CoRR*, abs/2308.10792.

Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proceedings of the Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10197–10207.

Task	Instruction Data
Vulnerability Detection	Instruction: Detect whether the following code contains vulnerabilities. Input: {The vulnerable code in Figure 2} Output: 1
Vulnerability Localization	Instruction: Identify any security vulnerabilities in the following code, and specify the lines where they occur. Input: {The vulnerable code in Figure 2} Output: 9: b+= ff_subtitles_next_line (b);
Vulnerability Interpretation	Instruction: Analyze the following code from the perspective of whether it contains vulnerabilities. Input: {The vulnerable code in Figure 2} Output: The vulnerability in svg_probe arises from how b is incremented in line 9. This, in combination with the control flow in line 8 and the dataflow/boundary check in line 14, can lead to an infinite loop under certain conditions. This scenario highlights the importance of meticulously managing pointer arithmetic and loop conditions in low-level programming, particularly in functions processing external inputs.

Table 5: Instruction examples of different task

A Example of Multi-task Data

For the vulnerable code as shown in Figure 2, when only the prompt from **Step 1** in Section 3.3 is used, GPT-4's response is: *"Buffer Overrun: The function relies on the behavior of ff_subtitles_next_line(b). If this function fails to correctly handle the end of the buffer, it could result in b pointing beyond the end, causing a buffer overrun."* However, the response following the complete CoT-SV is: *"The vulnerability in svg_probe arises from how b is incremented in line 9. This, in combination with the control flow in line 8 and the dataflow/boundary check in line 14, can lead to an infinite loop under certain conditions. This scenario highlights the importance of meticulously managing pointer arithmetic and loop conditions in low-level programming, particularly in functions processing external inputs."* These results indicate that when GPT-4 directly analyzes a vulnerable code, its interpretation can be imprecise and biased, and misidentifies the example code as **buffer overrun**. In contrast, the vulnerability interpretation provided by CoT-SV is more accurate and align more closely with the CVE descriptions as illustrated in Figure 2, emphasizing the **infinite loop**. Based on the vulnerability line obtained from Figure 2 and the above-mentioned vulnerability interpretation, we present the instruction fine-tuning data examples for three tasks as shown in Table 5.

B Dataset details

We perform undersampling on non-vulnerability functions to ensure the numbers of vulnerable and non-vulnerable samples are balanced. For Devign, we utilize the standard partitions provided by the dataset to create training, validation, and test sets. Regarding DiverseVul, which does not provide standard partitions, we randomly split them into the training, validation, and test sets with an 8:1:1 ratio, ensuring a 1:1 ratio for both classes in each set. Subsequently, we concatenate the training, validation, and test sets of the two datasets to obtain the corresponding training and validation sets. The final mixed dataset used for training, validation contains 23,078 and 2,864 examples, respectively. In our work, Big-Vul, CVEfixes, ReVeal, and Juliet are solely used as testing data. We also perform data cleaning, deduplication, and ensure a 1:1 ratio between positive and negative samples for these datasets. For Big-Vul and Juliet, we randomly select 10% of the processed data for testing. As for ReVeal, due to its limited size, we utilize the entire dataset for testing. Finally, the DiverseVul, Devign, BigVul, CVEfixes, ReVeal, and Juliet used for testing respectively contain 1,532, 1,312, 1,170, 4,216, 2,028, and 3,152 examples.

C Baselines

Devign (Zhou et al., 2019). Devign is a classic method that utilizes GNNs for vulnerability detection. It extracts information from mutiple dimensions of the code, encoding it into a joint graph, and employs GGNN to learn hidden layer representations. It uses a convolutional module to extract features from nodes for graph-level classification. Another significant contribution of Devign is the release of a large dataset collcted and manually labeled from 4 popular C language libraries. This dataset has been widely used in subsequent related works. In our implementation, we use the source code released in ReVeal (Chakraborty et al., 2022)

to conduct our experiments.

ReVeal (Chakraborty et al., 2022). Addressing issues such as data repetition and imbalanced data samples in existing datasets, ReVeal introduced a dataset constructed through its own collection efforts and conducted a systematic evaluation on this dataset. Additionally, ReVeal proposed a new vulnerability detection method. It represents code as a Code Property Graph (CPG), utilizes GGNN to obtain a graph representation, and then feeds it into a Multi-Layer Perceptron (MLP) layer for vulnerability detection.

CodeBERT (Feng et al., 2020). CodeBERT is a pre-trained model that is based on the RoBERTa model architecture, specifically designed for understanding and generating programming languages. Its training data consists of both programming languages (PL) and natural languages (NL), employing masked language modeling (MLM) and replaced token detection (RTD) as pre-training tasks. For fine-tuning CodePTMs on vulnerability detection, we adopte the parameter settings in CodeXGLUE (Lu et al., 2021).

GraphCodeBERT (Guo et al., 2021). GraphCodeBERT extends the BERT architecture. In addition to the Masked Language Modeling pre-training task on both natural language and code language inputs, GraphCodeBERT allows the incorporation of the structural information of the code (i.e., dataflow). Correspondingly, it introduces two additional pre-training tasks: edge prediction and node alignment. The edge prediction and node alignment tasks are designed to encourage the model to learn semantic relationships between code structures and mapping relationships between code tokens and variable representations.

UniXcoder (Guo et al., 2022). UniXcoder is an unified and cross-modal pre-trained programming language model based on a N-layer Transformer architecture. The model takes a code representation which enhanced by code comments and serialized Abstract Syntax Tree as input. UniXcoder utilizes self-attention masks to control the model's behavior between Encoder-Only, Decoder-Only, and Encoder-Decoder. It concurrently employs language modeling tasks corresponding to these three behaviors for pre-training the model. Additionally, the authors introduced two pre-training tasks to learn code semantic embeddings: multi-modal contrastive learning and cross-modal generation.

ReGVD (Nguyen et al., 2022). ReGVD is an effective model for code vulnerability detection. It treats source code as token sequences to construct graphs with node features initialized by a pre-trained language model. By leveraging GNNs with residual connections, ReGVD enhances learning and representation capabilities. The model combines sum and max pooling for graph embedding, which is then processed through a fully-connected and softmax layer to predict vulnerabilities.

EPVD (Zhang et al., 2023c). EPVD works by decomposing a code snippet into several execution paths, analyzing these paths using a CodePTM and a convolutional neural network (CNN) to capture both intra- and inter-path attention, and then combining these analyses to form a comprehensive code representation. This representation is then used by a multilayer perceptron (MLP) classifier to identify vulnerabilities. This method effectively addresses issues related to irrelevant information and long code snippets in traditional vulnerability detection approaches.

D Metrics

Precision (P) is the proportion of vulnerable code correctly predicted as vulnerability among all code predicted as vulnerability. **Recall** (R) is the proportion of vulnerable code correctly predicted as vulnerability among all known real vulnerable code. F1 denotes the harmonic mean of precision and recall and is calculated as: $F1 = 2*(P*R)/(P+R)$. Given that the F1 score represents the harmonic mean of precision and recall, it effectively balances the impact of both metrics. Utilizing the F_1 score allows for a more comprehensive and equitable evaluation of the performance of vulnerability detection models. This ensures that the system neither generates excessive false positives nor overlooks too many genuine vulnerabilities. Consequently, in all evaluations, the F_1 score is employed to evaluate the performance of different models.

E Implementation Details

All the experiments are conducted on an Ubuntu 20.04 server with AMD Ryzen Threadripper 3960X 24-Core Processor CPU, 128GB of RAM, and 2 NVIDIA A800 80G GPUs. For fine-tuning CodePTMs, the learning rate is set to 2e-5, the max length is set to 512, the batch size is set to 32, and the epoch is set to 5. These parameter settings are consistent with those established on the CodeXGLUE (Lu et al., 2021) benchmark. For fine-tuning Llama-2 and CodeLlama, the learning

rate is set to 1e-4, the max length also set at 512, the batch size is set to 32, and and the epoch is set to 3. For fine-tuning StarCoder, the learning rate is set to 2e-5, the max length also set at 512, the batch size is set to 16, and and the epoch is set to 3. To improve training efficiency, we load all LLMs with 8-bit quantization. We employ LoRA (Hu et al., 2022) for instruction-tuning LLMs. The specific settings for LoRA include: the rank is 16, the alpha value is set to 32. The target modules for Llama-2 and CodeLlama are set to 'q_proj', 'v_proj', 'k_proj', and 'o_proj', while the target module for StarCoder is set to 'c_proj', 'c_attn', 'q_attn'. The partial parameters of different LLMs vary due to the distinct model settings provided in the official code. We have adhered to these settings in our experiments.

F Adversarial Attack

MHM (Zhang et al., 2020). MHM utilizes an iterative identifier substitution method based on the Metropolis-Hastings (M-H) sampling (Metropolis et al., 1953). This attack involves randomly choosing potential replacements for local variables and then making a strategic decision to either accept or reject these substitutions. MHM's effectiveness in selecting adversarial examples is enhanced by utilizing both the predicted labels and their confidence scores from the target model.

WIR-Random (Zeng et al., 2022). WIR-Random employs the Word Importance Rank (WIR) method to establish the order in which identifiers are substituted. This attack assigns a rank to each identifier based on the change in probabilities produced by the model when the identifier is renamed to "UNK". Following this ranking, WIR-Random systematically substitutes the identifiers, choosing replacements from a random pool of candidates.

Dead Code Insertion. We employ a form of dead code construction in DIP (Na et al., 2023) as follows: `char var_2[] = "snippet";` Where `var` is an identifier randomly selected from the dataset. To avoid the low probability of duplicate names, it is named `var_2`. The code snippet is also randomly obtained from the dataset. An example of a dead code snippet is: `char xpath_2[] = "err = sock_do_ioctl(net, sock, cmd, (unsigned long)&ktv);"`. We ensure the syntactic correctness and semantic consistency of the original code while inserting the generated dead code into random positions within the code.