Noname manuscript No.
(will be inserted by the editor)

Learning-based Models for Vulnerability Detection: An Extensive Study

Chao Ni · Xin Yin · Liyu Shen · Xiaodan Xu · Shaohua Wang

Received: date / Accepted: date

Abstract Though many deep learning-based models have made great progress in vulnerability detection, we have no good understanding of these models, which limits the further advancement of model capability, understanding of the mechanism of model detection, and efficiency and safety of practical application of models. In this paper, we extensively and comprehensively investigate two types of state-of-the-art learning-based approaches (sequence-based and graph-based) by conducting experiments on a recently built large-scale dataset. We investigate seven research questions from five dimensions, namely *model capabilities*, *model interpretation*, *model stability*, *ease of use of model*, and *model economy*. We experimentally demonstrate the priority of sequence-based models and the limited abilities of both LLMs (e.g. ChatGPT and CodeLlama)

Both Chao Ni and Xin Yin contributed equally to this research.
Chao Ni is the corresponding author.
He is also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

Chao Ni
The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
E-mail: chaoni@zju.edu.cn

Xin Yin
The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
E-mail: xyin@zju.edu.cn

Liyu Shen
The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
E-mail: liyushen@zju.edu.cn

Xiaodan Xu
The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
E-mail: xiaodanxu@zju.edu.cn

Shaohua Wang
Central University of Finance and Economics, China
E-mail: davidshwang@ieee.org

and graph-based models. We explore the types of vulnerability that learning-based models skilled in and reveal the instability of the models though the input is subtlely semantical-equivalently changed. We empirically explain what the models have learned. We summarize the pre-processing as well as requirements for easily using the models. Finally, we initially provide vital information for the economical and safe practical use of these models.

Keywords Vulnerability Detection · Empirical Study · Deep Learning

1 Introduction

Automated vulnerability detection is a critical challenge in system security, and recent advances in learning-based approaches have demonstrated significant promise in addressing this problem. Notably, several studies have demonstrated that deep learning (DL)-based approaches can achieve remarkable performance in vulnerability detection, with accuracy rates reaching up to 95% (Li et al., 2018, 2021c; Russell et al., 2018; Li et al., 2017; Maiorca and Biggio, 2019; Suarez-Tangil et al., 2017; Zhou et al., 2019) and F1-scores as high as 90% (Fu and Tantithamthavorn, 2022; Song et al., 2022).

Despite the remarkable success of DL-based models in detecting vulnerabilities, our understanding of these models remains limited. For example, whether they can be used effectively and reliably in detecting real-world or most dangerous vulnerabilities, what kind of vulnerability the models are skilled in detecting, what kind of features these models have learned, whether the models can stably perform on semantically equal functions, what is the complexity and what is the cost if we want to use DL-based models, whether it will damage our privacy. Figuring out the answers to these questions can help us better develop and apply the models in practical usage, especially in the era of Large Language Models (LLMs).

Several studies (Steenhoek et al., 2023; Chakraborty et al., 2021; Yin et al., 2024a) have investigated the characteristics and capabilities of DL-based models for vulnerability detection; however, there are still notable limitations that hinder practical application: (1) The conclusions drawn by different works may be inconsistent. For instance, some studies (Wang et al., 2023; Wen et al., 2023) conclude that graph-based models outperform sequence-based models, while others (Fu and Tantithamthavorn, 2022; Ni et al., 2023) report the opposite observation. (2) Existing research primarily focuses on either graph-based or sequence-based models, neglecting a comprehensive examination of prominent LLMs (e.g., ChatGPT (ope, 2022) and CodeLlama (Roziere et al., 2023)), which may lead to biased conclusions. (3) Previous studies have largely concentrated on the capabilities and interpretability of DL-based models, overlooking important factors that affect users, such as the ease of use and the associated costs of employing these models.

In this paper, we systematically and comprehensively investigate the characteristics of several SOTA DL-based vulnerability detection models (Zhou et al., 2019; Chakraborty et al., 2021; Li et al., 2021a; Fu and Tantithamthavorn, 2022; Ni et al., 2023). We formulate research questions aimed at gaining a deeper understanding of these models, with the goal of distilling lessons and guidelines for improved practical usage. Our primary focus is on graph-based and sequence-based vulnerability detection models at the function level. To the best of our knowledge, this is the first paper to systematically investigate SOTA DL-based models across a wide range of aspects in the era of LLMs.

Table 1: Insights and Takeaways: Evaluation on Extensive Newly Built Dataset (MegaVul)

Dimension	Findings or Insights
Capability	① . Sequence-based models outperform graph-based models. ② . Different models have their own advantages in detecting different types of vulnerabilities. ③ . Sequence-based models are skilled in "Input Validation". ④ . Graph-based models are skilled in "API Abuse", "Input Validation" and "Security Features". ⑤ . Sequence-based models especially SVulD its promising potential in practical usage. ⑥ . LLMs under the prompt engineering are not yet competent for vulnerability detection and different prompts enable varying ability.
Interpretation	⑦ . Both graph-based and sequence-based methods focus on two types of statements: "Function Calls" and "Field Expressions". ⑧ . Learning-based models still have a limited ability to distinguish vulnerable functions from non-vulnerable functions. ⑨ . Feeding external called function information sequence-based method could further improve sequence-based models' ability.
Stability	⑩ . All the learning-based models are unstable to input changes even if these changes are semantically equivalent. ⑪ . Sequence-based models perform more stably than graph-based models.
Ease of Use	⑫ . Sequence-based models: easy to deploy, limited input size, fine-tuning, open-source, privacy-safe. ChatGPT: easy to use, larger input size; privacy-unsafe. ⑬ . Graph-based models: complete function, complex configurations, limited input size, fine-tuning, open-source, privacy-safe.
Economy	⑭ . Graph-based models need large amounts of time for data preprocessing, but they typically train and infer fast. Sequence-based models do not involve data preprocessing, with a comparable training time and longer inference time. ⑮ . ChatGPT is the most economical solution.

In particular, we conduct seven research questions and classify them into the following dimensions: **D1: model capabilities**, **D2: model interpretation**, **D3: model stability**, **D4: ease of use of model** and **D5: model economy**. More precisely, our first goal is to understand the capabilities of the learning models for vulnerability detection tasks, especially aiming at asking the following research questions:

- **RQ-1**: How do learning-based approaches perform on vulnerability detection? What are the variabilities across different models?
- **RQ-2**: What types of vulnerabilities are learning-based approaches skilled in detecting?
- **RQ-3**: Are Large Language Models capable of detecting vulnerabilities?

Our second study aims at the model interpretation. We adopt the state-of-the-art explanation tools to investigate what the model has learned as follows:

- **RQ-4**: What source code information does the learning-based model focus on? Do different types of learning-based models agree on similar important code features?

Our third study targets the stability of the studied learning-based models by investigating the impacts of semantically equivalent subtle modifications to their input.

- **RQ-5**: Do learning-based models agree on the vulnerability detection results with themselves when the input is insignificantly changed?

Our fourth study focuses on the ease of use by investigating the various efforts to build an effective model.

- **RQ-6**: What types of efforts should be paid before using a model? In what scenarios can learning-based models be applied?

Finally, our study focuses on the economy. We want to investigate the cost when adopting the models to detect vulnerability.

- **RQ-7**: What are the costs caused by models from both time and economic aspects?

To answer the aforementioned research questions, we investigate two types of SOTA DL-based models. These models used different deep learning architectures (e.g., transformer (Vaswani et al., 2017) or graph (Zhou et al., 2020)). Besides, to extensively and comprehensively analyze the models' ability, we conduct experiments on the recently built dataset named MageVul (Ni et al., 2024), which contains real-world projects' vulnerabilities by crawling more newly discovered vulnerabilities. Then, we carefully design experiments to discover the findings by answering seven RQs. Eventually, the main contribution of our work is summarized as follows and takeaway findings are shown in Table 1.

- We conduct an extensive comparison of learning-based approaches for vulnerability detection, including LLMs."
- We design seven RQs grouped into five important dimensions to understand learning-based approaches comprehensively.
- We release our reproduction package for further study: `https://github.com/vinci-grape/Learning-based-Models-for-VD`.

2 Experimental Setup

In this section, we first introduce our studied dataset. Following that, we briefly describe the studied learning-based models and evaluation metrics. Finally, the implementation details are presented.

2.1 Studied Dataset

Though many vulnerability-related datasets have been proposed, there are still some limitations that impact the verification of proposed models, including (1) *unreal vulnerability* (i.e., SARD (sar, 2018) is artificially synthesized), (2)

unreal data distribution (i.e., balanced distribution in Devign (Zhou et al., 2019)), (3) *limited diversity* (i.e., limited projects and vulnerability types in ReVeal (Chakraborty et al., 2021)), (4) *limited newly disclosed vulnerabilities* (i.e., no updated to Big-Vul (Fan et al., 2020) covering the period only from 2003 to 2019), and (5) *low-quality of dataset* (i.e., incomplete function, erroneously merged functions, missed commit message in Big-Vul (Fan et al., 2020)).

To address the issues above, recently, Ni et al. (Ni et al., 2024) built a large-scale, high-quality, data-rich, multi-dimensional C/C++ and Java dataset named MegaVul by crawling data from more open-source repositories, adopting sophisticated filtering strategies to improve the quality, and employing advanced techniques to extract complete functions. In addition to collecting the raw functions, MegaVul also provides more dimension information on the function, including the nine types of granularity abstraction of functions (i.e., *FUNC*, *VAR*, *STRING*, etc.), various types of function representations (i.e., *AST*, *PDG*), and the details of function modifications (i.e., *diff*). In summary, MageVul collects 17 Git-based code hosting platforms from 349 websites that had referenced the CVEs more than 100 times. The web-based code hosting platforms can be categorized into five main categories: GitHub, GitLab, GitWeb, CGit, and Gitiles. Considering both the regular updates (i.e., update every six months) and the stability of MegaVul, we consider the C/C++ dataset released in October 2023 for experiments. That is, for the C/C++ version, MegaVul contains 8,334 commits from 198,994 CVEs. Table 2 presents the statistical information of MegaVul and more details can be referred to their original work (Ni et al., 2024).

2.2 Studied Baselines and Evaluation Metrics

Baselines. To comprehensively compare the performance of existing work, in this paper, we consider the five state-of-the-art learning-based software vulnerability detection approaches and these approaches can be further divided into two finer categories: graph-based ones and sequence-based ones. The former group contains three methods (i.e., Devign (Zhou et al., 2019), Reveal (Chakraborty et al., 2021) and IVDetect (Li et al., 2021a)) and they transform source code into a graph to complexly represent its semantic. The latter group contains two approaches (i.e., LineVul (Fu and Tantithamthavorn, 2022) and SVulD (Ni et al., 2023)) and they treat the source code as the sequence of tokens to simply represent its semantic. Here, we briefly introduce these methods to make our paper self-contained.

Evaluation Metrics. To comprehensively investigate the performance of learning-based models for vulnerability detection, we adopt the following widely used evaluation metrics (Ni et al., 2022a, 2023; Zhou et al., 2019; Fu and Tantithamthavorn, 2022): Accuracy (A), Precision (P), Recall (R), and F1-score (F1).

Table 2: The statistics of MegaVul (C/C++)

Attributes	**MegaVul**
Number of Projects	736
Number of CVE IDs	5,714
Date range of crawled CVEs	2013/01∼2023/04 (continuously updating)
Number of CWE IDs	159
Number of Commits	6,437
Number of Crawled Code Hosting Platforms	17
Number of Vul/Non-Vul Function	14,216/377,185
Function Extract Strategy	Tree-sitter
Dimensions of Information	6
Code Availability	Full

2.3 Implementation Details

Data Splitting. Similar to existing work (Fu and Tantithamthavorn, 2022; Li et al., 2021a), we adopt the same data splitting approach: 80%:10%:10%. More precisely, the whole dataset is split into 80% of training data, 10% of validation data, and 10% of testing data. Meanwhile, we also keep the class distribution as same as the original ones in training data, validation data, and testing data.

Model Implementation. Regarding ReVeal, IVDetect, LineVul, and SVulD, we utilize their publicly available source code and perform fine-tuning with the default parameters provided in their original code. Considering Devign's code is not publicly available, we make every effort to replicate its functionality and achieve similar results on the original paper's dataset. All these models are implemented using the PyTorch (Paszke et al., 2019) framework by fully adopting the pre-trained models hosted on Hugging Face (hug, 2024). Additionally, we incorporate interpretability into all studied models. The fine-tuning process is performed on NVIDIA RTX 3090 graphics card. We utilize CodeLlama-7B, DeepSeek-Coder-6.7B, and Magicoder-6.7B for the setup of fine-tuning. In contrast, we leverage state-of-the-art models (i.e., ChatGPT, CodeLlama-34B, and DeepSeek-Coder-33B) for the setup of prompt engineering. We set the number of few-shot learning examples between 1 and 6 to fill the context window (i.e. 4,096 tokens).

3 Research Question and Findings

In this section, we divide our seven research questions into five dimensions: *D1: model capabilities*, *D2: model interpretation*, *D3: model stability*, *ease of use of model* and *model economy*. For each RQ, we introduce the objective, the experimental setup, the results, and our findings.

3.1 D1: Capabilities of Learning-based Models for Vulnerability Detection

•[RQ-1]: How do learning-based approaches perform on vulnerability detection? What are the variabilities across different models?

Objective. Many deep learning-based vulnerability detection approaches have been proposed (Zhou et al., 2019; Li et al., 2021a; Ni et al., 2023; Hanif and Maffeis, 2022; Li et al., 2018; Cao et al., 2022; Wang et al., 2023) and they mainly focus on function-level vulnerability detection, treating the source code in different ways. That is, some approaches (Zhou et al., 2019; Li et al., 2021a) consider the complex structure inside a function and transform it into a graph, while some approaches (Fu and Tantithamthavorn, 2022; Ni et al., 2023) simply treat it as a sequence of tokens without considering its structure (i.e., sequence-based). Though these methods have been well compared in previous studies (Steenhoek et al., 2023; Ni et al., 2023; Wang et al., 2023), their experiments are usually conducted on a limited or small-scale dataset, which may impact the consistency of models' capabilities. For example, Wen et al. (Wen et al., 2023) concluded that a complex graph-based model embedding a function by considering the program structure can yield better performance than sequence-based models. However, according to recent works (Ni et al., 2023; Fu and Tantithamthavorn, 2022), sequence-based models seem to outperform graph-based ones. Meanwhile, recently LLMs (especially ChatGPT) have attracted much attention since their powerful ability can be easily adapted to various types of downstream tasks, including vulnerability detection. However, there are no comparisons between LLMs and existing models. Considering these issues, we want to conduct an extensive study to comprehensively compare learning-based models' abilities.

Experimental Setup. We consider three graph-based approaches (i.e., Devign (Zhou et al., 2019), Reveal (Chakraborty et al., 2021) and IVDetect (Li et al., 2021a)) and two sequence-based approaches (i.e., LineVul (Fu and Tantithamthavorn, 2022) and SVulD (Ni et al., 2023)). Meanwhile, to comprehensively compare the performance difference, we adopt the currently largest dataset MageVul. Since graph-based approaches usually need to obtain the structure information of the function (e.g., CFG, DFG), we adopt the same toolkit with Joern to transform functions and drop the fail-passed cases. Finally, the filtered dataset (391,401, shown in Table 2) is used for evaluation and we follow previous work (Fu and Tantithamthavorn, 2022; Ni et al., 2022b) to split the dataset into the training data (i.e., 80%), validating data (i.e., 10%), and testing data (i.e., 10%). We also keep the distribution as same as the original ones in training, validating, and testing data.

Furthermore, we also consider the LLMs (i.e., CodeLlama (Roziere et al., 2023), DeepSeek-Coder (AI, 2023), Magicoder (Wei et al., 2023), and ChatGPT (OpenAI, 2022)) and they also treat source code as a sequence of tokens. Regarding CodeLlama, DeepSeek-Coder, and Magicoder, we perform discriminative fine-tuning utilizing the "AutoModelForSequenceClassification"

class provided by the Transformers library to implement discriminative fine-tuning. "AutoModelForSequenceClassification" is a generic model class that will be instantiated as one of the sequence classification model classes of the library when created with the "AutoModelForSequenceClassification.from pretrained(model name or path)" class method. We prompt ChatGPT with an in-context learning setting and equip it with 1∼6 examples selected from the same projects (cf. Section 3.1 for details). ChatGPT is a commercial conversation-based LLM model developed by OpenAI and can only be accessed by its API or web interface. Considering the large-scale testing size (i.e., 38,749 functions) as well as the substantial cost when interacting with ChatGPT, we follow previous work (Croft et al., 2023) to statistically sample some cases with 95% confidence and we conduct experiments on these sampled functions.

Table 3: The comparisons among learning-based approaches

Types	Models	Accuracy	Recall	Precision	F1-score
Graph Based	Devign	0.742	0.622	0.068	0.122
	Reveal	0.780	0.545	0.070	0.125
	IVDetect	0.792	0.582	0.080	0.141
Sequence Based	LineVul	**0.962**	0.593	**0.117**	**0.195**
	SVulD	0.822	**0.637**	0.100	0.172
	CodeLlama	0.836	0.525	0.093	0.158
	DeepSeek-Coder	0.844	0.533	0.099	0.167
	Magicoder	0.836	0.498	0.089	0.151
	ChatGPT*	0.932 ±0.015	0.125 ±0.020	0.057 ±0.014	0.078 ±0.016

*Notice that the performance of ChatGPT is calculated on statistical sampling with 95% confidence.

Results. Table 3 shows the comparison results and the best ones are highlighted in bold. From the results, we can draw the following observations: (1) Surprisingly, the sequence-based modes perform better than the graph-based models in terms of all evaluated metrics, which indicates that we may not be concerned about the complex code structure when utilizing deep learning techniques to build a vulnerability detector. (2) Among sequence-based models, these methods have a complementary ability to detect vulnerabilities. More precisely, LineVul performs better in terms of *Accuracy*, *Precision*, and *F1-score*, while SVulD performs better in terms of *Recall*. It means that sequence-based models can be used for different usage scenarios, for example, LineVul for high *Precision* and SVulD for high *Recall*. (3) The performance of fine-tuned LLMs (i.e., CodeLlama, DeepSeek-Coder, and Magicoder) is comparable to graph-based models; however, they exhibit inferior performance relative to existing sequence-based models (i.e., LineVul and SVulD) when evaluated using *F1-score*, *Precision*, and *Accuracy*. Considering the computational resources and time costs of deploying LLMs, existing sequence-based models for vulnerability detection are a more efficient choice. (4) Though ChatGPT's performance is obtained on the statistically sampled dataset with 95% confidence, its performance is still far away from the existing SOTA baselines, especially in

terms of *Recall*, *Accuracy* and *F1-score*, which means that currently, ChatGPT is not yet competent for vulnerability detection tasks. (5) Considering the original goal difference between ChatGPT (i.e., target various tasks including QA, NLP, SE, etc.) and existing sequence-based models (i.e., LineVul and SVulD, target exclusively vulnerability detection), we find that it is necessary to build a vulnerability detection targeted model to make further progress in the field.

Finding 1: (1) Sequence-based vulnerability detection models achieve better performance than graph-based models. (2) LLMs are not yet competent for software vulnerability detection and it is necessary to build a vulnerability detection targeted sequence-based model.

•[RQ-2]: What types of vulnerabilities are learning-based approaches skilled in detecting?

Objective. Many types of models have been proposed for vulnerability detection and among them, graph-based and sequence-based are the promising ones. However, different approaches may have their own advantages in detecting different types of vulnerabilities. Figuring out their expertise can better guide us in practical usage. Therefore, we want to analyze what are the types of vulnerabilities that each learning-based approach skilled in.

Table 4: Seven Types of Vulnerability

Vulnerability Type	# Total	# Testing	CWE Example
Input Validation and Representation	2,887	294	CWE-20
Code Quality	1,543	170	CWE-416
Security Features	376	32	CWE-284
API Abuse	17	4	CWE-252
Time and State	13	1	CWE-367
Errors	7	1	CWE-388
Encapsulation	-	-	CWE-501

*No C/C++ instance in MegaVul belongs to "Encapsulation".

Experimental Setup. We make an analysis of each approach's performance on vulnerability types in the testing dataset and pick up the Top-10 vulnerability types that are most correctly classified for each method. Besides, following previous work (Tsipenyuk et al., 2005), we can group the vulnerabilities into 7 categories, namely *Input Validation and Representation*, *API Abuse*, *Security Features*, *Time and State*, *Errors*, *Code Quality*, and *Encapsulation*, shown under vulnerability types in Table 4. Specifically, "Input Validation and Representation" is caused by metacharacters, alternate encodings, and numeric representations. The mapping of the complete CWE list to these groups can be found in our dataset. "API Abuse" is commonly caused by the caller failing to honor the end of a contract between the caller and callee, e.g.,

CWE-252 "Unchecked Return Value". "Security Features" mainly concerns topics like authentication, access control, confidentiality, cryptography, and privilege management, e.g., CWE 359 "Privacy Violation". "Time and State" mainly concerns the time and state in distributed computation for more than one component to communicate correctly by sharing the state and time, e.g., CWE-833 "Deadlock". "Errors" relates to a class of API that handles errors, e.g., CWE-1069 "Empty Exception Block". "Code Quality" mainly concerns the unpredictable behavior caused by poor code quality. It leads to poor usability for a user and provides an opportunity to stress the system in unexpected ways for an attacker, e.g., CWE-476 "NULL Pointer Dereference". "Encapsulation" aims to draw strong boundaries of operations, e.g., CWE-501 "Trust Boundary Violation". Notice that there are no instances in *Encapsulation* and no method can correctly detect vulnerability belonging to both "Errors" and "Time and State", we do not need to analyze them further.

Besides, we also analyze the Top-25 Most Dangerous Software Weaknesses[1] to figure out the promising approach in detecting the most dangerous vulnerabilities in practice. Notice that six of the most dangerous CWEs (i.e., CWE-352, CWE-434, CWE-502, CWE-77, CWE-798, and CWE-306) are not included in the testing dataset, we, therefore, delete them from the list. All the results are from the ones in RQ-1.

Table 5: Top-10 correctly detected CWE by each method

Approach	Top-1	Top-2	Top-3	Top-4	Top-5
Devign	CWE-78 [2/2]	CWE-918 [2/2]	CWE-276 [2/2]	CWE-863 [3/4]	CWE-94 [3/4]
Reveal	CWE-94 [4/4]	CWE-79 [2/2]	CWE-918 [2/2]	CWE-89 [1/1]	CWE-287 [3/4]
IVDetect	CWE-89 [1/1]	CWE-863 [3/4]	CWE-94 [3/4]	CWE-20 [61/86]	CWE-119 [104/148]
LineVul	CWE-918 [2/2]	CWE-89 [1/1]	CWE-863 [3/4]	CWE-20 [63/86]	CWE-119 [107/148]
SVulD	CWE-79 [2/2]	CWE-918 [2/2]	CWE-190 [29/35]	CWE-787 [65/79]	CWE-863 [3/4]
ChatGPT	CWE-476 [1/3]	CWE-125 [1/3]	CWE-787 [1/5]	/	/
Approach	**Top-6**	**Top-7**	**Top-8**	**Top-9**	**Top-10**
Devign	CWE-269 [3/4]	CWE-287 [3/4]	CWE-787 [58/79]	CWE-125 [47/70]	CWE-190 [23/35]
Reveal	CWE-787 [56/79]	CWE-20 [53/86]	CWE-190 [21/35]	CWE-125 [41/70]	CWE-119 [85/148]
IVDetect	CWE-787 [53/79]	CWE-190 [23/35]	CWE-125 [41/70]	CWE-476 [33/62]	CWE-362 [22/43]
LineVul	CWE-787 [56/79]	CWE-190 [23/35]	CWE-125 [44/70]	CWE-22 [4/7]	CWE-362 [24/43]
SVulD	CWE-287 [3/4]	CWE-119 [104/148]	CWE-20 [59/86]	CWE-362 [28/43]	CWE-125 [42/70]
ChatGPT	/	/	/	/	/

Results. Table 5 shows the results of the Top 10 CWE that each method performs well and Fig. 1 shows the performance of different vulnerability groups. From the results, we observe that: (1) Each method performs variously on different vulnerability types which indicates their complementary ability. (2) Overall, the most of methods perform best in "Input Validation and Representation" and perform relatively worst in "API Abuse". (3) Sequence-based approaches (e.g., LineVul and SVulD) perform similarly, but graph-based approaches perform differently. (4) ChatGPT performs poorly in all studied types of vulnerabilities.

[1] https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

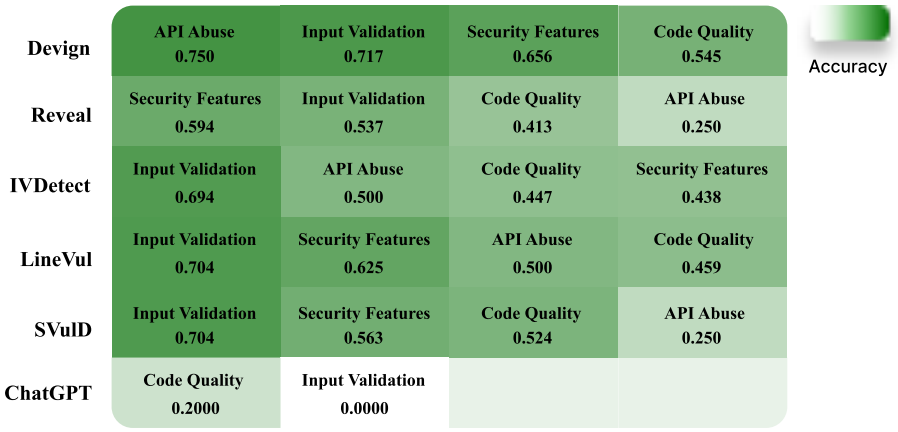


Fig. 1: Performance on Vulnerability Type

Table 6 shows the performance difference of each method on Top-25 most dangerous CWE. By observing these results, we conclude that: (1) Overall, sequence-based models perform better, especially SVulD, which shows their potentiality in practical usage. (2) As for graph-based models, Devign (i.e., 397) outperforms Reveal (i.e., 302) and IVDetect (i.e., 380) with an improvement of 95 and 17 functions correctly classified, respectively. As for the Top-5 dangerous CWEs, Devign also performs better, which shows the priority among other graph-based models. (3) As for sequence-based models, SVulD (i.e., 415) performs best and improves LineVul (i.e., 394) by 21. The powerful ability of SVulD is also consistent in the results of Top-5 dangerous CWEs.

Finding 2: (1) Different models have their own advantages in detecting different types of vulnerabilities. Particularly, sequence-based models are skilled in "Input Validation", but graph-based models have a wide range ("API Abuse", "Input Validation" and "Security Features"). (2) Generally, sequence-based models especially SVulD perform better and SVulD shows its promising potentiality in practical usage when detecting the most dangerous vulnerabilities.

•[RQ-3]: Are Large Language Models capable of detecting vulnerabilities under the prompt engineering?

Objective. Large Language Models (LLMs) (Brown et al., 2020) have been widely adopted since the advances in Natural Language Processing (NLP) which enable LLM to be well-trained with both billions of parameters and billions of training samples, resulting in substantial performance improvements across various tasks. LLMs can be easily used for a downstream task by being prompted (Liu et al., 2023; Yin, 2024) and they can capture different knowledge from various domain data. Previous studies (Liu et al., 2021; Lu et al., 2021; Yin et al., 2024b) have shown that the strength of LLMs may vary widely depending on the prompts. Therefore, we aim to investigate how LLMs perform

Table 6: The performance comparison among studied six approaches on Top-25 most risk CWE

ID	CWE	Graph-based			Sequence-based		
		Devign	**Reveal**	**IVdetect**	**LineVul**	**SVulD**	**ChatGPT***
1	CWE-787	53/79	41/79	53/79	56/79	**67/79**	1/5
2	CWE-79	**2/2**	1/2	0/2	1/2	1/2	0/0
3	CWE-89	**1/1**	**1/1**	**1/1**	**1/1**	0/1	0/0
4	CWE-416	34/72	26/72	26/72	31/72	**35/72**	0/2
5	CWE-78	0/2	0/2	0/2	**1/2**	**1/2**	0/0
6	CWE-20	56/86	43/86	61/86	**63/86**	61/86	0/4
7	CWE-125	**47/70**	41/70	41/70	44/70	41/70	1/3
8	CWE-22	**5/7**	4/7	3/7	4/7	4/7	0/0
11	CWE-862	0/1	0/1	0/1	0/1	0/1	0/0
12	CWE-476	34/62	27/62	33/62	30/62	**37/62**	1/3
13	CWE-287	3/4	**4/4**	2/4	2/4	3/4	0/0
14	CWE-190	27/35	23/35	23/35	23/35	**29/35**	0/1
17	CWE-119	100/148	70/148	105/148	**107/148**	103/148	0/0
19	CWE-918	**2/2**	0/2	1/2	**2/2**	**2/2**	0/0
21	CWE-362	**24/43**	16/43	22/43	**24/43**	**24/43**	0/0
22	CWE-269	**3/4**	2/4	2/4	1/4	2/4	0/0
23	CWE-94	**3/4**	2/4	3/4	0/4	2/4	0/0
24	CWE-863	1/4	1/4	**3/4**	**3/4**	**3/4**	0/0
25	CWE-276	**2/2**	0/2	1/2	1/2	0/2	0/0
# Wins (628)		**397**	**302**	**380**	**394**	**415**	3/18

*Notice that the performance of ChatGPT is calculated on statistical sampling with 95% confidence.

in detecting vulnerabilities across different prompt settings since no study has been conducted comprehensively on this topic.

Experimental Setup. We conduct experiments with the state-of-the-art LLMs (e.g., CodeLlama-34B (Roziere et al., 2023), DeepSeek-Coder-33B (AI, 2023), and ChatGPT (OpenAI, 2022)). Besides, considering the consumption of interaction with ChatGPT caused by the large-scale dataset (i.e., 38,749 functions), we statistically sample from the testing dataset as suggested by previous work (Croft et al., 2023), which can also reflect the target dataset as precise as possible. In particular, we sample the instances with 95% confidence and 3% interval[2]. Eventually, we obtain 1,039 instances to conduct our study. For CodeLlama-34B and DeepSeek-Coder-33B, we conduct experiments on the complete testing dataset (cf. Section 3.1 for details).

Meanwhile, considering that different prompts will affect the performance of LLMs in vulnerability detection, we adopt three prompt settings for our study: (1) **Zero-Shot:** which directly prompts LLMs to detect vulnerabilities without providing any demonstrations, (2) **In-Context-Learning (ICL):** enables LLMs to directly generate an answer for vulnerability detection task by

[2] https://surveysystem.com/sscalc.htm

feeding a few prompted demonstrations (i.e. a few shots) as part of the input, and (3) **Chain-of-Thought (CoT):** prompts LLMs to achieve an answer after a step-by-step process, which largely improves performance on reasoning. Studies have shown that CoT reasoning can be performed with zero-shot prompting (Zero-Shot CoT) (Kojima et al., 2022) or few-shot demonstrations (Few-Shot CoT) (Wei et al., 2022). We consider five distinct strategies for selecting demonstrations to explore the influence of different demonstrations on Few-shot ICL and Few-shot CoT. The details of the five selection strategies are elaborated as follows.

- **Fixed Selection.** We select pre-set fixed demonstrations in a sequential order from up to six CWEs (i.e., CWE-416, CWE-476, CWE-79, CWE-200, CWE-20 and CWE-787) until limitation are reached. These CWEs are selected from the Top-25 Most Dangerous Software Weaknesses.

- **Random Selection.** We randomly select a few demonstrations from training data (i.e., cf. Section 3.1 for details).

- **Random$_{repo}$ Selection**. We randomly select demonstrations from training data and these demonstrations are from the same projects that the target function belongs to.

- **Diversity-based Selection.** We adopt a pre-trained model (i.e., CodeBERT (Feng et al., 2020b)) to embed all the functions from training data and then use K-means algorithm (MacQueen et al., 1967) for clustering with six centers. The demonstrations that are closest to each cluster center are selected to ensure diversity.

- **Semantic-based Selection.** We utilize CodeBERT to embed all the functions from the training data as well as the target function. Subsequently, we select the most semantically similar demonstrations to the target function based on cosine similarity.

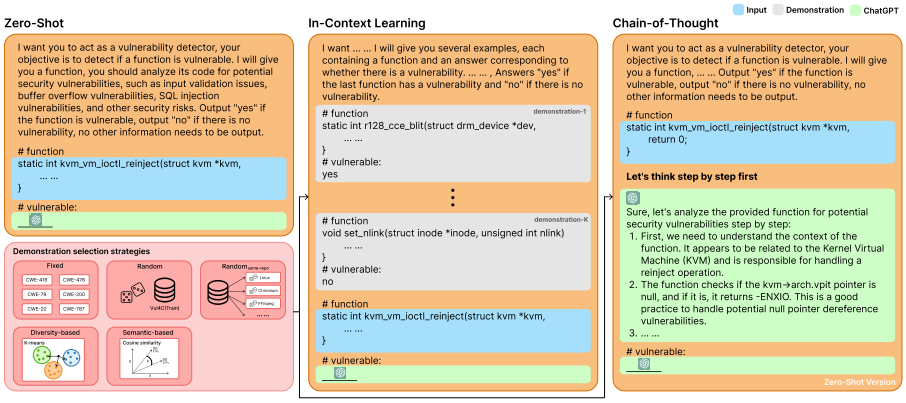


Fig. 2: Vulnerability detection prompt templates used in our study

Fig. 2 presents examples of templates under three different prompt settings. These prompt templates start with the instruction ``I want you to act as a vulnerability detector. Your objective is to detect... Output 'yes' if the function is vulnerable..." which explicitly states the task to be completed by the LLM and the expected output format. If the prompt includes additional demonstrations (i.e., in few-shot setting), ``I will give you several examples..." will also be inserted into the instruction to explicitly indicate to the LLM the presence of multiple functions and their vulnerability detection results within the prompt. In the few-shot setting, we employ the effective and efficient selection strategy mentioned above to choose as many demonstrations as possible from the training set until we reach the LLM's maximum input window token limitation (i.e., 4,096). Including more demonstrations in the prompt can convey task-specific knowledge to LLMs through the correlation between input and output (Min et al., 2022), thus enhancing LLM performance. More specifically, in the ICL setting, we employ five selection strategies. For CoT, we manually craft the reasoning process for each demonstration, and hand-crafted reasoning is superior to LLM-generated reasoning (Kojima et al., 2022). Therefore, we only consider the Few-Shot setting combined with two selection strategies (i.e., Fixed Selection and Diversity-based Selection). In the Zero-Shot Cot setting, we use ``Let's think by think" to prompt the LLM to generate its own reasoning process. These demonstrations and reasonings are incorporated into the prompt in a specific format, as denoted by the gray background in the figure. Subsequently, the function to be detected for vulnerabilities is added to the end of the template, forming the resulting prompt that instructs LLM to produce the final detection result.

Overall, we explore nine prompt designs for LLMs when detecting vulnerability and the details of the setting are illustrated in Table 7, Table 8, and Table 9.

Results. Table 7, Table 8, and Table 9 show the comparison results among different prompt designs for LLMs when detecting vulnerability. From the detailed results, we can achieve the following observations: (1) Different prompt settings result in varying performances and no one setting can achieve the best performance for all metrics. (2) Overall, for both ChatGPT and CodeLlama, the combination of CoT and Diversity-based Selection achieved the best performance in terms of *Recall* (i.e., 0.375 in ChatGPT and 0.444 in CodeLlama), improving other setting. (3) ChatGPT prompted with in-context learning as well as $Random_{repo}$ strategy performance well in terms of *Precision* (i.e., 0.057) and *F1* (i.e., 0.078) and also achieve a performance of *Recall* (i.e., 0.125). Meanwhile, CodeLlama and DeepSeek-Coder achieve the best performance with in-context learning as well as *Diversity-based* strategy. It seems to indicate that demonstrations from some domains with target functions may help LLMs better address similar tasks. (4) For ChatGPT, though "Zero-Shot" achieves best in terms of *Accuracy*, it fully performs worst in terms of the other three metrics. We further analyze and find that in this setting, ChatGPT almost predicts all functions as a clean one, It seems to

have no ability to distinguish between clean and vulnerable ones, which is also confirmed by its performance on the other three performance metrics. (5) Considering the highly imbalanced dataset in practice (i.e., 3.6% vulnerability in our dataset), CodeLlama prompted with in-context learning as well as *Diversity-based* strategy is the best setting.

Table 7: The prompt design for ChatGPT when detecting vulnerability

ZoSt	ICL	CoT	Example Selection Strategy					Accuracy	Recall	Precision	F1
			Fixed	Rdm	Rdm_{repo}	Div	Sem				
✓								**0.977**	0	0	0
	✓		✓					0.960	0.042	0.050	0.046
	✓			✓				0.934	0.042	0.021	0.028
	✓				✓			0.932	0.125	**0.057**	**0.078**
	✓					✓		0.921	0.042	0.017	0.024
	✓						✓	0.946	0.042	0.029	0.035
✓		✓						0.867	0.083	0.017	0.028
		✓	✓					0.961	0	0	0
		✓				✓		0.733	**0.375**	0.033	0.061

*"ZoSt": Zero-Shot; "ICL": In-Context Learning; "CoT": Chains-of-Thoughts; "Rdm": Random; "Div.": Diversity; "Sem": Semantic

Table 8: The prompt design for CodeLlama when detecting vulnerability

ZoSt	ICL	CoT	Example Selection Strategy					Accuracy	Recall	Precision	F1
			Fixed	Rdm	Rdm_{repo}	Div	Sem				
✓								0.661	0.222	0.018	0.033
	✓		✓					0.732	0.222	0.023	0.041
	✓			✓				0.699	0.222	0.020	0.037
	✓				✓			0.719	0.259	0.025	0.046
	✓					✓		0.732	0.407	**0.040**	**0.073**
	✓						✓	0.713	0.333	0.031	0.057
✓		✓						0.701	0.407	0.036	0.066
		✓	✓					**0.790**	0.296	0.039	0.068
		✓				✓		0.694	**0.444**	0.038	0.070

Table 9: The prompt design for DeepSeek-Coder when detecting vulnerability

ZoSt	ICL	CoT	Example Selection Strategy					Accuracy	Recall	Precision	F1
			Fixed	Rdm	Rdm_{repo}	Div	Sem				
✓								0.450	**0.593**	0.028	0.053
	✓		✓					0.655	0.370	0.028	0.053
	✓			✓				0.766	0.222	0.026	0.047
	✓				✓			0.693	0.185	0.017	0.030
	✓					✓		**0.804**	0.259	**0.037**	0.064
	✓						✓	0.672	0.444	0.036	0.066
✓		✓						0.598	0.444	0.029	0.054
		✓	✓					0.648	0.407	0.031	0.057
		✓				✓		0.650	0.482	0.036	**0.067**

Finding 3: (1) LLMs have limited ability to be directly used to detect the vulnerability and different prompt designs will highly affect their performance. (2) Overall, ChatGPT prompted with in-context learning as well as Random_{repo} selection strategy performs the best in terms of *Precision* and *F1-score*.

3.2 D2: Interpretation of Learning-based Models for Vulnerability Detection

• **[RQ-4]: What source code information does the learning-based model focus on? Do different types of learning models agree on similar important code features?**

Objective. Vulnerability detection models should help developers understand how they make their predictions (i.e., identify vulnerable code patterns). Therefore, it is meaningful to investigate whether different deep learning models make decisions based on specific types of statements and help the model better be understood. For instance, the model might pay more attention to "if" statements when detecting input validation vulnerabilities.

Additionally, different types of learning-based approaches (i.e., graph-based and sequence-based) may focus on varying types of information, and figuring out the difference of code features concerned by models can help to better improve their abilities.

Experimental Setup. To explain the types of statements the model focuses on, we need to obtain the score for each token in the source code. We employ different interpretability techniques to acquire precise scores of tokens based on the characteristics of the studied models. For graph-based models (e.g., Devign (Zhou et al., 2019) and IVDetect (Li et al., 2021a)), we utilize GNNExplainer (Ying et al., 2019), which provides scores for each edge in the constructed graph, and subsequently, we calculate the score for each node by aggregating the scores of all incoming edges. Besides, since a node may contain several tokens, we assign the score to each corresponding token. As for the Reveal, it employs a two-stage architecture consisting of a GNN for learning feature vectors and a representation model for classification. We adopt DeepLift (Shrikumar et al., 2019) for the representation model to unveil the contribution of each neuron to the final prediction. For sequence-based models (e.g., LineVul (Fu and Tantithamthavorn, 2022) and SVulD (Ni et al., 2023)), we use the attention layer to get each tokens' score since they are Transformer-based model (Vaswani et al., 2023), naturally providing reasoning behind the prediction decision (Serrano and Smith, 2019).

After obtaining scores for each token, we can obtain the score for each line by summing up the scores of all tokens within it. For each correctly classified vulnerable function in the testing dataset, we select the top 10 lines with the highest scores and treat them as the most important code features contributing to the model's decision. Subsequently, we utilize Tree-sitter (tre, 2024) to parse

Table 10: Types of Statements

Statement Type	**Brief Description**
If Statement	*if* keyword and condition expression
For Statement	*for* keyword, initialization, condition, iteration expression
While Statement	*while* keyword and condition expression
Jump Statement	*goto*, *break*, *continue*
Switch Statement	*switch* keyword and condition expression
Case Statement	*case*,*default* keyword and value expression
Return Statement	*return* keyword
Arithmetic Operation	+, -, *, /, % Binary expression
Relational Operation	==, !=, ¿, ¡, ¿=, ¡= Binary expression
Logical Operation	&& , || Binary expression
Bitwise Operation	&, |, ∧ , <<, >> Binary expression
Declaration Statement	variable type and name

15 types of statements (shown in Table 10) within the functions and count the occurrences of each statement type among the top 10 lines.

We also apply t-SNE (van der Maaten and Hinton, 2008), a visualization technology mapping high-dimensional features into two-dimensional features, to explore the separability of studied models between vulnerable functions and non-vulnerable functions. For a better illustration, we randomly select 10,000 examples from the testing dataset and extract the hidden vectors before making the final binary classification decision as the high-dimensional features for different models, e.g., the hidden vector of the `[CLS]` used for sequenced-based models and the hidden features of each node used for graph-based models.

Results. Fig. 3 shows the results and we obtain the following findings: (1) "Function Call" and "Field Expression" are the most risky operations identified by both graph-based and sequenced-based models. The operating frequency of the two statement types exceeds 50% among the studied 15 different statement types, which seems that both operations will introduce unstable factors to functionality and are prone to introduce vulnerabilities. For example, Fig. 4 shows a function from the Linux project that aims to parse the channel attribute in the WIFI configuration. The code (Line 21) makes a function call (i.e., "*le16_to_cpu*") and brings a risk to the current function. That is, the external function should not be called directly for another operation, and further input validation is required to ensure the attribute has enough space to avoid "*out-of-bounds write*" vulnerability. (2) The models exhibit relatively low attention towards "for", "case", "while", "jump", and "switch" statement types. We conducted a manual analysis of these functions and found that the statements are generally simple, making them less prone to vulnerabilities. For example, the "*while*" condition statement (Line 13 in Fig. 4) is straightforwardly presented, and developers can easily identify the termination criteria while writing the program. (3) Sequence-based models (i.e., LineVul and SVulD) perform similarly on different types of statements, possibly because both of

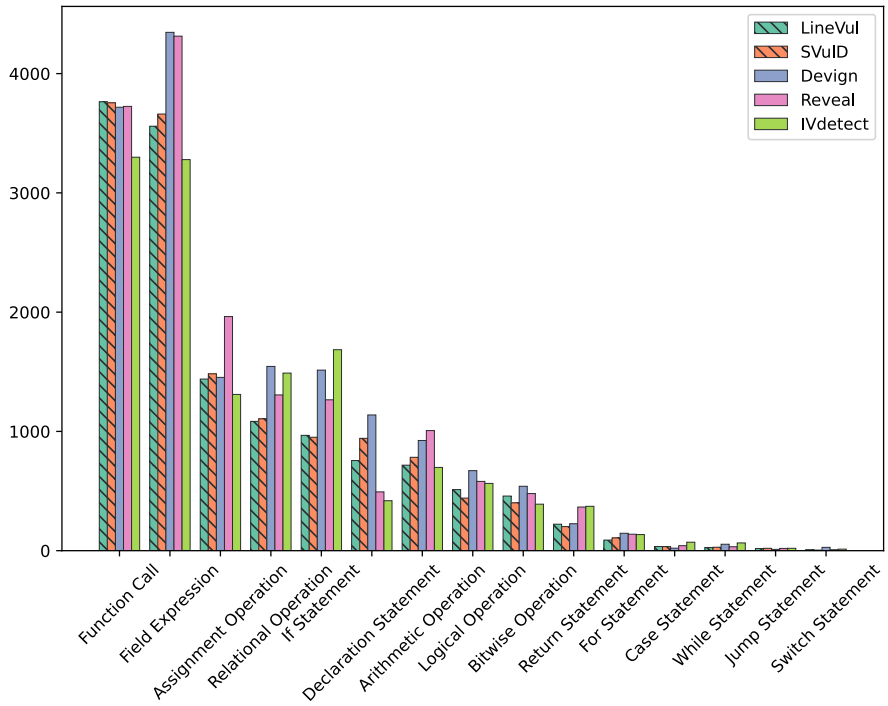


Fig. 3: Number of occurrences for each statement type in the Top 10 most probable vulnerability lines (diagonal shadow indicates sequence-base models)

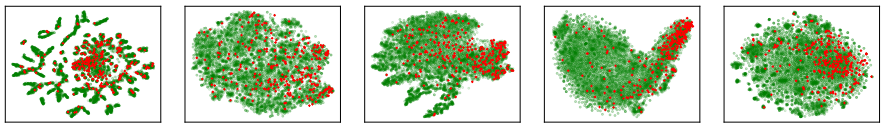
them are built upon CodeBERT (Feng et al., 2020a) and variants (Guo et al., 2022). For example, for each type of statement, the two methods seem to achieve the same attention numbers. (4) Graph-based models pay varying attention to different types of statements. For example, for "Field Expression", both Reveal and Devign pay more attention than IVDetect. For "Assignment Operation", the Reveal pays more attention than both Devign and IVDetect. The difference may be caused by the way to encode the internal node among graph-based models. IVDetect strives to encode as much information as possible from a single line of code (e.g., AST, CDG, etc.) into a single node, Devign directly utilizes nodes generated by *Joern* as the nodes presented in the graph, which may explain why IVDetect shows less sensitive to the operation of accessing or operating members in *class* or *struct*, since IVDetect merges multiple field expressions into a single node, losing the detailed information, and consequently reduces its attention to such statement type.

Fig. 5 illustrates the visualization of separating vulnerable functions from non-vulnerable functions and we obtain the following observations: (1) All the figures show an overlap between the functions with or without vulnerabilities, which means that all the learning-based models have limited ability to distinguish them. By analyzing the types of statements that the models focus

```
1  static inline void wilc_wfi_cfg_parse_ch_attr(u8 ......
2  {
     ......
13   while (index + sizeof(*e) <= len) {
14     e = (struct wilc_attr_entry *)&buf[index];
15     if (e->attr_type == IEEE80211_P2P_ATTR_CHANNEL_LIST)
16       ch_list_idx = index;
17     else if (e->attr_type == IEEE80211_P2P_ATTR_OPER_CHANNEL)
18       op_ch_idx = index;
19     if (ch_list_idx && op_ch_idx)
20       break;
21     index += le16_to_cpu(e->attr_len) + sizeof(*e);
22   }
23
24   if (ch_list_idx) {
       ......
46 }
```

Fig. 4: LineVul interpretation result (CVE-2022-47519)

on (i.e., "Function Call"), it seems that the models need the context of the externally called functions to enrich the input information, which helps to better understand the functionality. (2) Sequence-based models (i.e., LineVul and SVulD) seem to have a better separation boundary (i.e., more concentrated) than graph-based models, especially LineVul seems to perform best, which is also consistent with the results obtained in RQ-1.



(a) Devign (b) Reveal (c) IVDetect (d) LineVul (e) SVulD

Fig. 5: Visualization of the separation between vulnerable (denoted by +) and non-vulnerable (denoted by ◯)

Finding 4: (1) Both graph-based and sequence-based methods technically focus on two types of statements: Function Calls and Field Expressions, which may involve vulnerable or incredible operations to functionality. (2) The existing learning-based models still have limited ability to distinguish vulnerable functions from non-vulnerable functions. Sequence-based models perform better than the graph-based models. (3) Feeding external called function information sequence-based method could further improve sequence-based models' ability.

• **[RQ-5]: Do learning-based models agree on the vulnerability detection results with themselves when the input is insignificantly changed?**

Objective. An optimal vulnerability detection model should base its decisions on the root cause of vulnerabilities while demonstrating robustness against the potential impact of code layout or unrelated noise. Therefore, we want to assess the stability of the studied models by evaluating their generalizability to slightly modified but semantically equivalent input.

Experimental Setup. We apply four types of semantic-preserving transformations (introduction along with examples are shown in Table 11) to each testing sample in the original MegaVul dataset to construct four distinct variants of the test set. **(1) Remove all comments.** For each function, we remove all the comments in its source code. **(2) Insert comments.** For each function, we randomly insert single-line comments into its function body. The number of inserted comments equals 15% of the total number of lines in the function body, and the insertion positions are randomly selected. Note that the content of the comments may not be relevant to the function. **(3) Insert irrelevant code.** We randomly insert a single line of unrelated code for each function, which will not influence the function's functionality and output. **(4) Rename all identifiers.** For each function, we consistently replace the names of its parameters and variables declared within its function body with `VAR0`, `VAR1`, $\cdots$, `VARX`, which ensures that the program semantics and functionalities remain unchanged. Then, we test the models trained in Section 3.1 on the four variant test sets. We adopt a comprehensive performance metric F1 to analyze the performance difference.

Table 11: Semantic-preserving Transformation Types

Transformation Type	Summary	Example
Remove all comments	Remove all comments from the function	~~/* Initializing variables before the main loop. */~~
Insert comments	Randomly insert comments into the function	/* A loop to iterate over elements in an array. */
Insert irrelevant code	Randomly insert unrelated code into the function	if(0) {}
Rename all identifiers	Replace parameters and declared variables with VARX	int delta; ➨ int VAR2;

Results. Table 12 shows the results of the studied models on the original test set and its four variants. From the results, we achieve the following observations. (1) All studied models are unstable to the four types of semantic-preserving transformations. (2) Sequence-based models achieve a relatively smaller performance change, which means that these models are more stable than graph-based models. The robustness of the sequence-based models may be explained by their elaborate and complex model architectures with large

amounts of parameters and also indicates the existence of less meaningful tokens in code (Zhang et al., 2022). (3) Graph-based models suffer from a severe performance decrease. In particular, IVDetect is affected the most, whose performance drops by 61.0%∼82.3%. (4) Overall, "Rename all identifiers" impacts more to models' stability than other types of transformations.

Table 12: Performance difference between original MegaVul test set and its four semantically-equivalent variants

Types	Models	Original	Remove All Comments	Insert Comments	Insert Irrelevant Code	Rename All Identifiers
Graph Based	Devign	0.122	0.075 (38.5%↓)	0.075 (38.5%↓)	0.073 (40.2%↓)	0.075 (38.5%↓)
	Reveal	0.125	0.099 (20.8%↓)	0.093 (25.6%↓)	0.094 (24.8%↓)	0.097 (22.4%↓)
	IVDetect	0.141	0.051 (63.8%↓)	0.052 (63.1%↓)	0.055 (61.0%↓)	0.025 (82.3%↓)
Sequence Based	LineVul	0.195	0.196 (0.5%↑)	0.186 (4.6%↓)	0.195 (-)	0.183 (6.2%↓)
	SVulD	0.172	0.174 (1.2%↑)	0.163 (5.2%↓)	0.179 (4.1%↑)	0.186 (8.1%↑)
	CodeLlama	0.158	0.155 (1.9%↓)	0.153 (3.2%↓)	0.155 (1.9%↓)	0.146 (7.6%↓)
	DeepSeek-Coder	0.167	0.165 (1.2%↓)	0.160 (4.2%↓)	0.162 (3.0%↓)	0.151 (9.6%↓)
	Magicoder	0.151	0.147 (2.6%↓)	0.143 (5.3%↓)	0.145 (4.0%↓)	0.141 (6.6%↓)
	ChatGPT*	0.078	0.073 (6.4%↓)	0.068 (12.8%↓)	0.089 (14.1%↑)	0.066 (15.4%↓)

*Notice that the performance of ChatGPT is calculated on statistical sampling with 95% confidence.

Finding 5: All the learning-based models are unstable to input changes even if these changes are semantically equivalent. Sequence-based models are more stable to subtle input changes than graph-based models.

3.4 D4: Ease of Use of Learning-based Models for Vulnerability Detection

• [RQ-6]: What types of efforts should be paid before using a model? In what scenarios can learning-based models be applied?

Objective. We want to assess the ease of use of the vulnerability detection models by examining their input requirements and model features. These aspects can offer valuable insights for practitioners who seek practical applications of these models.

Experimental Setup. We carefully document the key steps for reproducing the graph-based, sequence-based models and LLMs. Specifically, we verify the input requirements for each model by examining its requirement of program integrity (i.e., whether it can handle incomplete input programs), compilation (i.e., whether the input program needs to be compiled), and input size (i.e., the upper limit of input). Furthermore, during training and inference, we record for each model whether it requires fine-tuning to ensure its optimal performance, whether its source code is available, the minimum hardware requirement, the configuration difficulty, and the data privacy security level.

Results. We summarize the ease of use of the models in Table 13. From the results, we obtain the following conclusions: (1) Graph-based models require complete input programs since their inputs must be successfully parsed into

Table 13: Ease of Use of Learning-based Models

Models	Input Requirements			Model Features				
	Program Integrity	Compilation	Input Size	Fine-Tuning	Code Availability	Hardware Requirement	Configuration Difficulty	Privacy
Devign	✓	✗	Medium	✓	✓	>1GB	Difficult	Safe
Reveal	✓	✗	Medium	✓	✓	>1GB	Difficult	Safe
IVDetect	✓	✗	Medium	✓	✓	>1GB	Difficult	Safe
LineVul	✗	✗	Small	✓	✓	>6GB	Medium	Safe
SVulD	✗	✗	Small	✓	✓	>6GB	Medium	Safe
CodeLlama	✗	✗	Large	✓	✓	>72GB	Easy	Safe
DeepSeek-Coder	✗	✗	Large	✓	✓	>72GB	Easy	Safe
Magicoder	✗	✗	Large	✓	✓	>72GB	Easy	Safe
ChatGPT	✗	✗	Large	✗	✗	API	Easy	Unsafe

graphs, while sequence-based models do not require program integrity. (2) None of the models require the input programs to be compilable. (3) ChatGPT has the largest input size (≤16K tokens), while sequence-based models LineVul and SVulD are limited to a small input size (≤ 512 tokens). The input size of graph-based models is medium. (4) All the models, except for ChatGPT, have released their implementation code and require fine-tuning to enhance the performance, while ChatGPT is closed-source and hard to fine-tune. (5) ChatGPT is the most user-friendly method, as it can be used directly through API or on a website. Graph-based models demand small memory (>1GB) but require complex preprocessing steps and configurations to construct code graphs, while sequence-based models require larger memory (>6GB) but involve only a small amount of coding work. (6) All models, except for ChatGPT, are privacy-safe as they can be deployed on the user's own server, while ChatGPT carries a potential risk of privacy leakage.

Finding 6: Graph-based models require complete input programs and complex configurations to construct code graphs, while sequence-based models are easier to deploy. Except for ChatGPT, all current models are relatively limited by input sizes, require fine-tuning to achieve enhanced performance, and are open-source and privacy-safe. ChatGPT is the most user-friendly option regarding input requirements and model configurations, but it presents a potential risk of privacy leakage.

3.5 D5: Economy Impact of Learning-based Models for Vulnerability Detection

•[RQ-7]: What are the costs caused by models from both time and economic aspects?

Objective. Deploying vulnerability models in a real-world setting requires appropriate resource allocation to ensure high cost-effectiveness. Users are often interested in factors such as the effort required for model training and deployment, the model's processing speed for incoming requests, and the budget associated with the deployment. Therefore, in this RQ, we aim to assess the time and economic costs of the models.

Experimental Setup. We conduct experiments on a server with a uniform configuration equipped with an Intel(R) Xeon(R) Platinum 8358P CPU @

2.60GHz, 755GB of RAM, and 10 NVIDIA GeForce RTX 3090 graphics cards. During the data preprocessing phase, we utilize the tools Glove, Word2Vec, and Joern. Glove and Word2Vec need to train on the train set, while Joern needs to extract graph information for all functions in the dataset. We adopt the latest versions of these tools available on GitHub. We decided to perform model training and inference on a single RTX 3090 graphics card and adjust the batch size to maximize the use of the GPU memory. We execute the experiments three times and calculate the average running time results to mitigate the bias. We use the API provided by PyTorch to iteratively obtain the parameter size of the models. The inference cost of using ChatGPT is calculated according to the pricing strategy provided on the OpenAI (ope, 2024) official website, and the version of ChatGPT we used is GPT-3.5 Turbo with a 4K context window. For the other deep learning models, we calculate the cost of going from preprocess data to inference whole test set on hourly pricing using AWS's g5.xlarge instance (aws, 2024), which utilizes an NVIDIA A10G with similar performance to the NVIDIA 3090.

Table 14: Time and economic costs of the models

Model	Time			Parameter	Cost
	Pre-processing	Training	Inference		
Devign	7,103s	2,836s	101s	0.97M	2.81$
Reveal	7,103s	5,220s	148s	1.09M	4.44$
IVDetect	3,563s	13,602s	916s	1.01M	4.88$
LineVul	0s	13,274s	322s	124.65M	6.17$
SVulD	0s	6,048s	319s	125.93M	1.78$
CodeLlama	0s	222,621s	1,570s	7B	352.23$
DeepSeek-Coder	0s	209,683s	1,359s	6.7B	332.51$
Magicoder	0s	208,529s	1,328s	6.7B	330.64$
ChatGPT	-	-	1,263s	-	0.64$

Results. The results are summarized in Table 14. Based on the results, we can obtain the following findings: (1) Graph-based models require a significant time cost for data preprocessing, sometimes exceeding the time needed for model training. IVDetect has the longest training time among graph-based models due to its complex model structure, where the input goes through multiple layers, such as TreeLSTM, GloVe, GNN, and the pooling layer. In contrast, the model architectures of Devign and Reveal are relatively simpler. In particular, Devign trains the fastest because it only adopts a single-layer GatedGraphConv. (2) Sequence-based models, especially LLMs, are more complex in structures with larger amounts of model parameters, which explains their longer inference time. However, sequence-based models also have advantages: they require zero data preprocessing time; LineVul and SVulD are comparable to graph-based models in training time. Though LineVul and SVulD have similar model structures (i.e., 12 transformer layers) and we set the epoch equally as 20, LineVul requires more training time because its official implementation not adopting an early stopping

mechanism. (3) Among all the models, ChatGPT is the most economical option with a cost of only 0.64$, which shows its potential for practical usage.

Finding 7: Graph-based models need large amounts of time for data preprocessing, but they typically train and infer fast. In contrast, sequence-based models do not involve data preprocessing, with a comparable training time and longer inference time. Overall, ChatGPT is the most economical solution.

4 Threats to Validity

Internal Validity arises from two aspects. The first one is about the uncertainty of LLM's output. Previous work has verified that LLMs are sensitive to prompts, such as the number and quality of selected examples in-context learning and chain-of-thoughts, and natural language instruction. To alleviate this threat, we explore the performance of different example strategies in RQ1 and use fixed instructions and random seeds to ensure the generated content is relatively consistent. In addition, ChatGPT is a closed-source LLM, which poses a threat to reproducibility, so the results we report may relate to a specific version of ChatGPT (i.e., GPT-3.5 Turbo). Another potential threat is the implementation of a graph-based vulnerability detection model. To mitigate this threat, we leverage the open-source implementations provided by previous works. In cases where the code is unavailable, we employ paired programming to ensure a close replication of the performance reported in the original paper. Furthermore, we strictly adhere to the hyperparameters reported in the original papers.

External Validity concerns the generalization of our report results. The first threat comes from the fact that we focus on vulnerability detection in C and C++ languages, many disclosed vulnerabilities in other popular languages (e.g., Java or Python) are not considered in this study. Another threat is the impact of dataset selection. To mitigate this threat, we have created the MegaVul dataset to cover most of the C/C++ vulnerabilities recorded in the NVD database since 2003, which is the largest function-level vulnerability dataset, ensuring that the evaluation results are representative and convincing.

5 Related Work

Vulnerability detection (VD) has attracted much attention and many learning-based approaches have been proposed to automatically learn the vulnerability patterns from historical data (Yamaguchi et al., 2014; Li et al., 2018; Zhou et al., 2019; Li et al., 2021b; Duan et al., 2019; Lin et al., 2017; Chakraborty et al., 2021; Li et al., 2021c). These methods can be further divided into complex graph-based ones (Yamaguchi et al., 2014; Zhou et al., 2019; Cheng et al., 2021; Wang et al., 2020; Cao et al., 2022; Hin et al., 2022) and sequence-based

ones (Dam et al., 2017; Russell et al., 2018; Fu and Tantithamthavorn, 2022; Ni et al., 2023), and have become state-of-the-art.

Recently, a few works have conducted empirical studies on these learning-based vulnerability detection models. Chakaborthy et al. (Chakraborty et al., 2021) investigated the issues of synthetic datasets, data duplication, and data imbalance by studying four deep learning models and then improved their model design based on their findings. Tang et al. (Tang et al., 2020) surveyed two models to investigate the best methods among neural network architectures, vector representation methods, and symbolization methods. Lin et al. (Lin et al., 2021) construct dataset including nine software projects to evaluate six neural network models' vulnerability detection ability and their generalization. Meanwhile, Ban et al. (Ban et al., 2019) evaluated six learning based models in a cross-project setting considering three software projects. Steenhoek et al. (Steenhoek et al., 2023) also conduct an empirical study on deep learning based vulnerability detection models with the consideration of three dimensions (i.e., model capabilities, training data, and model interpretation).

Different from these works, our work extensively studies the characteristics of learning-based VD approaches in the era of large pre-trained language models, especially focusing on ChatGPT's remarkable ability by considering five dimensions. To the best of our knowledge, our work is the first attempt to characterize the ChatGPT's ability on VD, the ease of use of models, the model economy, and types of vulnerability that models are skilled in.

6 Conclusion

This paper aims to comprehensively investigate the capabilities of graph-based and sequence-based learning-based models for vulnerability detection as well as their impacts. To achieve that, we first build a large-scale vulnerability dataset and then conduct several experiments focusing on five dimensions: *model capabilities*, *model interpretation*, *model stability*, *ease of use of model*, and *model economy*. The results indicate the priority of sequence-based models and the limited abilities of both LLMs and graph-based models. We also investigate the performance of learning-based models on types of vulnerability and find that both sequence-based and graph-based models are skilled in "Input Validation", while graph-based models are skilled at another two: "API Abuse" and "Security Feature". We also find that all learning-based models perform inconsistently. Finally, we conclude the pre-processing and requirements for easy usage of models and obtain vital information for economically and safely practical usage of these models.

References

(2018) Software assurance reference dataset (sard). `https://samate.nist.gov/SARD/`

(2022) Chatgpt: Optimizing language models for dialogue. URL https://chat.openai.com
(2024) Aws g5 instance. https://aws.amazon.com/cn/ec2/instance-types/g5/
(2024) Hugging face. URL https://huggingface.co
(2024) Openai pricing. https://openai.com/pricing
(2024) Tree-sitter. https://github.com/tree-sitter/tree-sitter
AI D (2023) Deepseek coder: Let the code write itself. https://github.com/deepseek-ai/DeepSeek-Coder
Ban X, Liu S, Chen C, Chua C (2019) A performance evaluation of deep-learnt features for software vulnerability detection. Concurrency and Computation: Practice and Experience 31(19):e5103
Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A, et al. (2020) Language models are few-shot learners. Advances in neural information processing systems 33:1877–1901
Cao S, Sun X, Bo L, Wu R, Li B, Tao C (2022) Mvd: Memory-related vulnerability detection based on flow-sensitive graph neural networks. arXiv preprint arXiv:220302660
Chakraborty S, Krishna R, Ding Y, Ray B (2021) Deep learning based vulnerability detection: Are we there yet. IEEE Transactions on Software Engineering
Cheng X, Wang H, Hua J, Xu G, Sui Y (2021) Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Transactions on Software Engineering and Methodology (TOSEM) 30(3):1–33
Croft R, Babar MA, Kholoosi MM (2023) Data quality for software vulnerability datasets. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, pp 121–133
Dam HK, Tran T, Pham T, Ng SW, Grundy J, Ghose A (2017) Automatic feature learning for vulnerability prediction. arXiv preprint arXiv:170802368
Duan X, Wu J, Ji S, Rui Z, Luo T, Yang M, Wu Y (2019) Vulsniper: Focus your attention to shoot fine-grained vulnerabilities. In: IJCAI, pp 4665–4671
Fan J, Li Y, Wang S, Nguyen TN (2020) A c/c++ code vulnerability dataset with code changes and cve summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp 508–512
Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M (2020a) CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020, Association for Computational Linguistics, Online, pp 1536–1547, DOI 10.18653/v1/2020.findings-emnlp.139
Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, et al. (2020b) Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:200208155
Fu M, Tantithamthavorn C (2022) Linevul: A transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories, pp 608–620

Guo D, Lu S, Duan N, Wang Y, Zhou M, Yin J (2022) Unixcoder: Unified cross-modal pre-training for code representation. arXiv preprint arXiv:220303850

Hanif H, Maffeis S (2022) Vulberta: Simplified source code pre-training for vulnerability detection. In: 2022 International joint conference on neural networks (IJCNN), IEEE, pp 1–8

Hin D, Kan A, Chen H, Babar MA (2022) Linevd: Statement-level vulnerability detection using graph neural networks. arXiv preprint arXiv:220305181

Kojima T, Gu SS, Reid M, Matsuo Y, Iwasawa Y (2022) Large language models are zero-shot reasoners. Advances in neural information processing systems 35:22199–22213

Li B, Roundy K, Gates C, Vorobeychik Y (2017) Large-scale identification of malicious singleton files. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, pp 227–238

Li Y, Wang S, Nguyen TN (2021a) Vulnerability detection with fine-grained interpretations. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 292–303

Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y (2018) Vuldeepecker: A deep learning-based system for vulnerability detection. In: Proceedings of the 25th Annual Network and Distributed System Security Symposium

Li Z, Zou D, Xu S, Chen Z, Zhu Y, Jin H (2021b) Vuldeelocator: a deep learning-based fine-grained vulnerability detector. IEEE Transactions on Dependable and Secure Computing

Li Z, Zou D, Xu S, Jin H, Zhu Y, Chen Z (2021c) Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing

Lin G, Zhang J, Luo W, Pan L, Xiang Y (2017) Poster: Vulnerability discovery with function representation learning from unlabeled projects. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp 2539–2541

Lin G, Xiao W, Zhang LY, Gao S, Tai Y, Zhang J (2021) Deep neural-based vulnerability discovery demystified: data, model and performance. Neural Computing and Applications 33(20):13287–13300

Liu J, Shen D, Zhang Y, Dolan B, Carin L, Chen W (2021) What makes good in-context examples for gpt-3? arXiv preprint arXiv:210106804

Liu P, Yuan W, Fu J, Jiang Z, Hayashi H, Neubig G (2023) Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM Computing Surveys 55(9):1–35

Lu Y, Bartolo M, Moore A, Riedel S, Stenetorp P (2021) Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. arXiv preprint arXiv:210408786

van der Maaten L, Hinton G (2008) Visualizing data using t-sne. Journal of Machine Learning Research 9(86):2579–2605, URL `http://jmlr.org/papers/v9/vandermaaten08a.html`

MacQueen J, et al. (1967) Some methods for classification and analysis of multivariate observations. In: Proceedings of the fifth Berkeley symposium

on mathematical statistics and probability, Oakland, CA, USA, vol 1, pp 281–297

Maiorca D, Biggio B (2019) Digital investigation of pdf files: Unveiling traces of embedded malware. IEEE Security & Privacy 17(1):63–71

Min S, Lyu X, Holtzman A, Artetxe M, Lewis M, Hajishirzi H, Zettlemoyer L (2022) Rethinking the role of demonstrations: What makes in-context learning work? In: Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, pp 11048–11064, DOI 10.18653/v1/2022.emnlp-main.759, URL https://aclanthology.org/2022.emnlp-main.759

Ni C, Wang W, Yang K, Xia X, Liu K, Lo D (2022a) The Best of Both Worlds: Integrating Semantic Features with Expert Features for Defect Prediction and Localization. In: Proceedings of the 2022 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, pp 672–683

Ni C, Yang K, Xia X, Lo D, Chen X, Yang X (2022b) Defect identification, categorization, and repair: Better together. arXiv preprint arXiv:220404856

Ni C, Yin X, Yang K, Zhao D, Xing Z, Xia X (2023) Distinguishing look-alike innocent and vulnerable code by subtle semantic representation learning and explanation. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 1611–1622

Ni C, Shen L, Yang X, Zhu Y, Wang S (2024) Megavul: A c/c++ vulnerability dataset with comprehensive code representation. In: Proceedings of 21th International Conference on Mining Software Repositories (MSR)

OpenAI (2022) Chatgpt: Optimizing language models for dialogue. (2022). https://openai.com/blog/chatgpt/

Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, et al. (2019) Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems 32

Roziere B, Gehring J, Gloeckle F, Sootla S, Gat I, Tan XE, Adi Y, Liu J, Sauvestre R, Remez T, et al. (2023) Code llama: Open foundation models for code. arXiv preprint arXiv:230812950

Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, Ellingwood P, McConley M (2018) Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE international conference on machine learning and applications (ICMLA), IEEE, pp 757–762

Serrano S, Smith NA (2019) Is attention interpretable? arXiv preprint arXiv:190603731

Shrikumar A, Greenside P, Kundaje A (2019) Learning important features through propagating activation differences. 1704.02685

Song Z, Wang J, Liu S, Fang Z, Yang K, et al. (2022) Hgvul: A code vulnerability detection method based on heterogeneous source-level intermediate representation. Security and Communication Networks 2022

Steenhoek B, Rahman MM, Jiles R, Le W (2023) An empirical study of deep learning models for vulnerability detection. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, pp 2237–2248

Suarez-Tangil G, Dash SK, Ahmadi M, Kinder J, Giacinto G, Cavallaro L (2017) Droidsieve: Fast and accurate classification of obfuscated android malware. In: Proceedings of the seventh ACM on conference on data and application security and privacy, pp 309–320

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

Tsipenyuk K, Chess B, McGraw G (2005) Seven pernicious kingdoms: A taxonomy of software security errors. IEEE Security & Privacy 3(6):81–84

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. Advances in neural information processing systems 30

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I (2023) Attention is all you need. 1706.03762

Wang H, Ye G, Tang Z, Tan SH, Huang S, Fang D, Feng Y, Bian L, Wang Z (2020) Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Transactions on Information Forensics and Security 16:1943–1958

Wang W, Nguyen TN, Wang S, Li Y, Zhang J, Yadavally A (2023) Deepvd: Toward class-separation features for neural network vulnerability detection. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, pp 2249–2261

Wei J, Wang X, Schuurmans D, Bosma M, Chi E, Le Q, Zhou D (2022) Chain of thought prompting elicits reasoning in large language models. arXiv preprint arXiv:220111903

Wei Y, Wang Z, Liu J, Ding Y, Zhang L (2023) Magicoder: Source code is all you need. arXiv preprint arXiv:231202120

Wen XC, Chen Y, Gao C, Zhang H, Zhang JM, Liao Q (2023) Vulnerability detection with graph simplification and enhanced graph representation learning. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, pp 2275–2286

Yamaguchi F, Golde N, Arp D, Rieck K (2014) Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy, IEEE, pp 590–604

Yin X (2024) Pros and cons! evaluating chatgpt on software vulnerability. arXiv preprint arXiv:240403994

Yin X, Ni C, Wang S (2024a) Multitask-based evaluation of open-source llm on software vulnerability. IEEE Transactions on Software Engineering

Yin X, Ni C, Wang S, Li Z, Zeng L, Yang X (2024b) Thinkrepair: Self-directed automated program repair. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp 1274–1286

Ying R, Bourgeois D, You J, Zitnik M, Leskovec J (2019) Gnnexplainer: Generating explanations for graph neural networks. 1903.03894

Zhang Z, Zhang H, Shen B, Gu X (2022) Diet code is healthy: Simplifying programs for pre-trained models of code. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 1073–1084

Zhou J, Cui G, Hu S, Zhang Z, Yang C, Liu Z, Wang L, Li C, Sun M (2020) Graph neural networks: A review of methods and applications. AI open 1:57–81

Zhou Y, Liu S, Siow J, Du X, Liu Y (2019) Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: In Proceedings of the 33rd International Conference on Neural Information Processing Systems, p 10197–10207