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

# Learning-based Models for Vulnerability Detection: An Extensive Study

**Chao Ni · Xin Yin · Liyu Shen · Shaohua Wang**

Received: date / Accepted: date

**Abstract** While deep learning-based models have achieved remarkable progress in vulnerability detection, our understanding of these models remains limited, which hinders further advancement in model capability, mechanistic understanding of detection processes, and efficient and safe practical deployment. This paper presents a comprehensive investigation of state-of-the-art learning-based models, including sequence-based models, graph-based models, and Large Language Models (LLMs), through extensive experiments conducted on MegaVul, a recently constructed large-scale vulnerability dataset. We systematically explore seven research questions across five critical dimensions: *model capability*, *model interpretation*, *model robustness*, *ease of model deployment*, and *model economy*. Our experimental findings reveal the superiority of sequence-based models over graph-based models and demonstrate the limited effectiveness of current LLMs (e.g., ChatGPT and CodeLlama) for vulnerability detection. We identify the specific vulnerability types that

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

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

different learning-based models excel at detecting and reveal the instability of the models through subtle semantic equivalent changes in the input. Through interpretability analysis, we provide empirical insights into what these models actually learn and focus on during the detection process. Additionally, we systematically summarize the pre-processing requirements and deployment considerations necessary for practical model usage. Finally, our study provides essential guidelines for the economical and safe practical application of learning-based models, offering valuable insights for both researchers and practitioners.

**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 models have demonstrated significant promise in addressing this problem. Notably, several studies have demonstrated that learning-based models 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 this remarkable success, our understanding of learning-based vulnerability detection models remains limited. Critical questions persist regarding their effectiveness in detecting real-world or most dangerous vulnerabilities, the specific vulnerability types they excel at detecting, the features they learn, their robustness on semantically equivalent functions, their deployment complexity and costs, and their privacy implications. Addressing these questions is crucial for enhancing model development and practical deployment, particularly 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 learning-based models for vulnerability detection; however, there are still notable limitations that hinder practical application: (1) **Inconsistent conclusions**: 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. (2) **Limited scope**: 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) **Narrow evaluation**: Previous studies concentrate on model capability and interpretability, overlooking practical factors such as ease of deployment and associated costs.

This paper systematically investigates the characteristics of state-of-the-art learning-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 to gain a deeper understanding of these models and distill practical guidelines for improved practical usage. Our focus encompasses graph-based (i.e., Devign (Zhou et al., 2019), Reveal (Chakraborty et al., 2021), and IVDetect (Li et al., 2021a)) and sequence-based (i.e., LineVul (Fu and Tantithamthavorn, 2022), SVulD (Ni et al., 2023), and LLMs (ope, 2022; Roziere et al., 2023; AI, 2023; Wei et al., 2023)) vulnerability detection models at the function level. To the best of our knowledge, this is the first systematic investigation of state-of-the-art learning-based models across comprehensive dimensions in the era of LLMs.

In particular, we conduct seven research questions and classify them into the following dimensions: **D1: model capability**, **D2: model interpretation**, **D3: model robustness**, **D4: ease of model deployment**, and **D5: model economy**. More precisely, our goal is to understand the capabilities of the learning-based models for vulnerability detection, especially by addressing the following research questions:

- **RQ-1**: How do learning-based models perform on vulnerability detection? What are the variabilities across different learning-based models?
- **RQ-2**: What types of vulnerabilities are learning-based models 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 do the learning-based models focus on? Do different types of learning-based models agree on similar important code features?

Our third study targets the robustness 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 learning-based models? 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 learning-based models to detect vulnerability.

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

**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.<br>② Different models have their own advantages in detecting different types of vulnerabilities.<br>③ Sequence-based models are skilled in "Input Validation".<br>④ Graph-based models are skilled in "Input Validation" and "API Abuse".<br>⑤ Sequence-based models, especially SVulD, show promising potential for practical usage.<br>⑥ LLMs with prompt engineering are not enough for vulnerability detection, and different prompts enable varying abilities. |
| **Interpretation** | ⑦ Both graph-based and sequence-based models focus on two types of statements: "Function Call" and "Field Expression".<br>⑧ Learning-based models still have a limited ability to distinguish vulnerable functions from non-vulnerable functions. |
| **Robustness** | ⑨ All the learning-based models are unstable to input changes, even if these changes are semantically equivalent.<br>⑩ Sequence-based models perform more robustly than graph-based models. |
| **Ease of Deployment** | ⑪ Sequence-based models are easier to deploy.<br>⑫ Except for ChatGPT, all models are open-source and privacy-safe. |
| **Economy** | ⑬ 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.<br>⑭ SVulD is the most economical solution. |

To answer the aforementioned research questions, we investigate two types of SOTA learning-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 MegaVul (Ni et al., 2024), which contains real-world projects' vulnerabilities by crawling 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 the takeaway findings are shown in Table 1.

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

## 2 Related Work

Vulnerability detection (VD) has attracted significant attention, with numerous learning-based models proposed to automatically learn 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; Ni et al., 2025). These models can be broadly categorized into graph-based models (Yamaguchi et al., 2014; Zhou et al., 2019; Cheng et al., 2021; Wang et al., 2020; Cao et al., 2022; Hin et al., 2022) that capture complex structural relationships within code, and sequence-based models (Dam et al., 2017; Russell et al., 2018; Fu and Tantithamthavorn, 2022; Ni et al., 2023) that treat code as token sequences.

Many vulnerability detection models lack explainability, leading to unreliable and opaque detection results that are difficult to trust in practice. To address this limitation, researchers have proposed interpretable detection methods that can be broadly categorized into two approaches. The first approach integrates interpretability directly into the detection model architecture. LineVul (Fu and Tantithamthavorn, 2022) provides statement-level interpretation of vulnerabilities by leveraging the self-attention mechanism of Transformer models to highlight important code segments. Similarly, LineVD (Hin et al., 2022) employs graph attention networks (GAT) to simultaneously perform detection and provide interpretation through attention weights. The second approach involves designing specialized interpretation frameworks that work alongside existing vulnerability detection models. Zou et al. (Zou et al., 2021) developed an explainable framework that identifies token combinations contributing significantly to predictions by systematically perturbing samples near the decision boundary. IVDetect (Li et al., 2021a) incorporates GNNExplainer (Ying et al., 2019) to provide post-hoc explanations for graph-based vulnerability detectors. Hu et al. (Hu et al., 2023) established principled guidelines for assessing the quality of interpretation approaches specifically designed for GNN-based vulnerability detectors, addressing key concerns in vulnerability detection practice.

Recently, several empirical studies have been conducted on learning-based vulnerability detection models. Chakraborty et al. (Chakraborty et al., 2021) investigated issues of synthetic datasets, data duplication, and data imbalance by studying four deep learning models and improved their model design based on their findings. Tang et al. (Tang et al., 2020) surveyed two models to investigate optimal methods among neural network architectures, vector representation methods, and symbolization approaches. Mazuera-Rozo et al. (Mazuera-Rozo et al., 2021) evaluated shallow and deep models on both binary classification and bug type classification tasks. Lin et al. (Lin et al., 2021) constructed datasets including nine software projects to evaluate six neural network models' vulnerability detection ability and generalization capabilities. Ban et al. (Ban et al., 2019) evaluated six learning-based models in cross-project settings across three software projects. Steenhoek et al. (Steenhoek et al., 2023) conducted an empirical study on deep learning-based vulnerability detection models considering three dimensions: model capabilities, training data, and model interpretation. Yin et al. (Yin et al., 2024a) performed an extensive technical evaluation of pre-trained models using Big-Vul across four software vulnerability tasks: vulnerability detection, assessment, location, and description.

However, despite these valuable contributions, several critical aspects remain underexplored in existing empirical studies:

- A comprehensive model comparison across different paradigms (i.e., graph-based, sequence-based, and LLMs) has been limited by small-scale datasets

and inconsistent experimental settings, leaving questions about their relative capabilities unresolved.

- Existing studies lack a vulnerability-type-specific analysis to understand which approaches excel at detecting particular types of vulnerabilities, limiting practical guidance for model selection.

- While LLMs like ChatGPT have shown remarkable abilities in various domains, a systematic evaluation of prompting strategies for vulnerability detection remains unexplored.

- Although interpretability methods exist, a comparative analysis of which types of statements various models focus on during detection has not been thoroughly investigated.

- Model stability and robustness to semantically equivalent but syntactically different code variants has received insufficient attention.

- Practical deployment considerations such as ease of deployment, input requirements, and model features have been largely overlooked in empirical evaluations.

- Economic and resource considerations for real-world deployment, including training costs, inference time, and budget requirements, have not been systematically studied.

Different from these works, our work extensively studies the characteristics of learning-based VD models in the era of LLMs by addressing these seven critical dimensions: (1) comprehensive model capability comparison across paradigms, (2) vulnerability-type-specific performance analysis, (3) systematic LLM prompting strategy evaluation, (4) comparative interpretability analysis, (5) model robustness assessment, (6) practical deployment considerations, and (7) economic and resource cost analysis. To the best of our knowledge, our work is the first comprehensive attempt to characterize LLMs' vulnerability detection capabilities, systematically evaluate prompting strategies, analyze model robustness, assess practical deployment factors, and conduct economic feasibility analysis in the context of learning-based vulnerability detection.

## 3 Experimental Setup

This section presents our experimental setup. We first introduce the dataset used for evaluation, followed by a description of the selected learning-based models and evaluation metrics. Finally, we detail the implementation specifics for reproducibility.

### 3.1 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., Big-Vul lacks updates beyond its 2003-2019 coverage), and (5) *low-quality of dataset* (i.e., incomplete function, erroneously merged functions, missed commit message in Big-Vul (Fan et al., 2020)). Croft et al. (Croft et al., 2023) conducted a comprehensive data quality analysis and systematically examined widely-used datasets, revealing that 20-71% of vulnerability labels were inaccurate in real-world datasets, and 17-99% of data points were duplicated. More recently, Ding et al. (Ding et al., 2024) conducted an extensive analysis of existing vulnerability datasets and found alarming label accuracy issues: BigVul achieved only 25% label accuracy for vulnerable functions, Devign had merely 24% accuracy, and significant data duplication with up to 18.9% test samples leaked from training sets. These findings challenge the reliability of models trained and evaluated on such datasets, suggesting that reported performance may be significantly overestimated.

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 light of these systematic studies revealing fundamental dataset quality issues, our choice of MegaVul represents a response to the documented limitations of existing datasets. Unlike previous datasets that suffer from low label accuracy and data leakage issues identified by Croft et al. and Ding et al., MegaVul employs more rigorous data collection and filtering strategies specifically designed to address these concerns. 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, MegaVul 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 6,437 commits related to 5,714 CVEs. Table 2 presents the statistical information of MegaVul, and more details can be referred to their original work (Ni et al., 2024).

**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 | 11,295/376,194 |
| Function Extract Strategy | Tree-sitter |
| Dimensions of Information | 6 |
| Code Availability | Full |

### 3.2 Studied Baselines and Evaluation Metrics

*Baselines.* To comprehensively evaluate vulnerability detection performance across different paradigms, we carefully select representative state-of-the-art learning-based models based on the following criteria: (1) *methodological diversity* - covering major architectural approaches, (2) *recency and impact* - focusing on recent influential works, (3) *reproducibility* - prioritizing models with available implementations, and (4) *performance competitiveness* - including top-performing models from recent literature (Fu and Tantithamthavorn, 2022; Yin et al., 2024a; Ni et al., 2025).

Based on these criteria, we consider nine prominent models that can be categorized into two groups:

**Graph-based models** include Devign (Zhou et al., 2019), Reveal (Chakraborty et al., 2021), and IVDetect (Li et al., 2021a). Devign is selected as a pioneering work that established graph neural networks for vulnerability detection with significant impact. Reveal represents advances in deep learning-based vulnerability detection with a comprehensive evaluation on real-world datasets. IVDetect is chosen for its novel integration of interpretability with detection performance.

**Sequence-based models** include LineVul (Fu and Tantithamthavorn, 2022) and SVulD (Ni et al., 2023). LineVul is selected as the current transformer-based approach, demonstrating superior performance across multiple datasets. SVulD represents the advancement in semantic representation learning for distinguishing subtle vulnerability patterns.

We also incorporate various LLMs, including CodeLlama (Roziere et al., 2023), DeepSeek-Coder (AI, 2023), Magicoder (Wei et al., 2023), and ChatGPT (OpenAI, 2022). These models are selected to represent different scales (6.7B to 34B parameters) and training paradigms, providing

comprehensive coverage of the current LLM landscape for vulnerability detection.

*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, Precision, Recall, and F1-score.

### 3.3 Implementation Details

*Data Splitting.* Since graph-based models require structural information of functions (e.g., CFG and DFG), we employ the Joern toolkit to transform functions and exclude failed parsing cases. The resulting filtered dataset containing 387,489 functions (detailed in Table 2) is used for evaluation. Following previous works (Fu and Tantithamthavorn, 2022; Ni et al., 2022b), we split the dataset into training (80%), validation (10%), and testing (10%) sets while maintaining the original vulnerability distribution across all splits.

For closed-source LLMs (i.e., CodeLlama, DeepSeek-Coder, and Magicoder), as the costs are manageable, we conduct experiments on the complete testing set. Considering the consumption of interaction with ChatGPT caused by the large-scale dataset (i.e., 38,749 functions in the testing set), we need to sample the testing set. To ensure our samples reflect the target testing set as precisely as possible, we statistically sample from the testing set as suggested by previous work (Croft et al., 2023). This approach allows us to determine the minimum number of instances needed to obtain datasets that are representative of the entire testing set. In particular, we sample the instances with 95% confidence and 3% interval[1]. Furthermore, we also maintain the original distribution of vulnerabilities (i.e., vul:non-vul ratio) found in the complete testing set, ensuring that our analysis on the sampled subset remains meaningful and generalizable to the broader set. Eventually, we obtain 1,039 instances to conduct our study. Subsequently, we conduct experiments on these sampled functions using ChatGPT.

*Model Implementation.* Regarding Reveal, IVDetect, LineVul, and SVulD, we utilize their publicly available source code and perform training/fine-tuning with the default parameters provided in their original implementations, such as 10 epochs, 2e-5 learning rate, and 16 training batch size for LineVul as specified in their original repository (Fu and Tantithamthavorn, 2022). 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).

For evaluation of LLMs, we utilize CodeLlama-7B, DeepSeek-Coder-6.7B, and Magicoder-6.7B for the setup of fine-tuning. We perform

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

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 "AutoModelForSequence Classification.from_pretrained(model_name_or_path)" class method. In contrast, we leverage ChatGPT (i.e., gpt-3.5-turbo-0125), 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). The fine-tuning and prompt engineering processes are performed on 10 × NVIDIA RTX 3090 GPUs. Parameter count constraints are imposed by our available computational resources.

## 4 Research Question and Findings

In this section, we divide our seven research questions into five dimensions: *D1: model capability*, *D2: model interpretation*, *D3: model robustness*, *D4: ease of model deployment*, and *D5: model economy*. For each RQ, we introduce the objective, the supplementary setting, the results, and our findings.

### 4.1 D1: Capability of Learning-based Models for Vulnerability Detection

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

**Objective.** Many deep learning-based vulnerability detection models 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 models (Zhou et al., 2019; Li et al., 2021a) consider the complex structure inside a function and transform it into a graph, while some models (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 models 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.

**Supplementary Setting.** For a more equitable performance comparison against other learning-based models, we specifically fine-tune three open-source LLMs: CodeLlama-7B, DeepSeek-Coder-6.7B, and Magicoder-6.7B. Since ChatGPT is a proprietary, conversation-based LLM developed by OpenAI that can only be accessed through its API or web interface, a different approach was necessary. Therefore, we prompt ChatGPT with an in-context learning setting and equip it with 1∼6 examples selected from the same projects (refer to Section 4.1 for details about the prompt and example selection).

**Table 3:** The comparisons among learning-based models

| 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 | 0.125 | 0.057 | 0.078 |

*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 models 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 models 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 (i.e., LineVul and SVulD) 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 model.

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

**Objective.** A diverse spectrum of learning-based models has been proposed for vulnerability detection, with graph-based and sequence-based models emerging as promising paradigms. While these models demonstrate varying performance across different vulnerability types, their specialized expertise remains underexplored. Understanding the specific vulnerability types that each model excels at detecting is crucial for informed model selection and deployment in real-world scenarios. To this end, we investigate the vulnerability detection capabilities of different learning-based models across various vulnerability types to identify their respective strengths and optimal application domains.

**Supplementary Setting.** To comprehensively assess the vulnerability detection capabilities of different models, we conduct our analysis from three complementary perspectives.

First, we analyze each model's performance on individual vulnerability types in the testing set, identifying the Top-10 vulnerability types that each model classifies most accurately.

Second, we examine model performance on the Top-25 Most Dangerous Software Weaknesses[2] to evaluate their effectiveness in detecting the most dangerous vulnerabilities in practice. Note that six CWEs from this list (i.e., CWE-352, CWE-434, CWE-502, CWE-77, CWE-798, and CWE-306) are not represented in our testing set and are therefore excluded from the analysis.

Third, to provide a systematic understanding of model strengths across broader vulnerability categories, we adopt the taxonomy proposed by Tsipenyuk et al. (2005), which groups 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,

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

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 model can correctly detect vulnerability belonging to both "Errors" and "Time and State", we do not need to analyze them further.

**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".

**Table 5:** Top-10 correctly detected CWE by each model

| **Model** | **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] | / | / |
| **Model** | **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*** | / | / | / | / | / |

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

**Table 6:** The performance comparison among six studied models on the 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.

**Results.** To provide a comprehensive understanding of model vulnerability detection capabilities, we present our findings from three complementary perspectives: individual vulnerability type analysis, the most dangerous vulnerability assessment, and categorical vulnerability evaluation.

**Individual Vulnerability Type Analysis.** Table 5 presents the Top-10 vulnerability types that each model detects most accurately. From the detailed analysis of the results, we observe that: (1) Each model demonstrates varying performance across different vulnerability types, indicating their complementary detection capabilities. (2) LineVul and SVulD show considerable overlap in their top-performing vulnerabilities, both excelling at CWE-918, CWE-863, CWE-787, CWE-119, CWE-20, CWE-190, CWE-125, and CWE-362, indicating similar detection capabilities. (3) The three graph-based models (i.e., Devign, Reveal, IVDetect) demonstrate more diverse patterns, with each model showing distinct strengths in different vulnerability categories.

**The Most Dangerous Vulnerability Assessment.** Table 6 examines model performance on the Top-25 most dangerous vulnerabilities in practice. From these results, we observe that: (1) Overall, sequence-based models perform better, especially SVulD, which shows their potential 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.

**Categorical Vulnerability Evaluation.** Fig. 1 illustrates model performance across the seven vulnerability categories. This categorical analysis provides several important insights: (1) Most models achieve their best performance in "Input Validation and Representation" vulnerabilities while performing relatively poorly in "API Abuse" categories. (2) Different model architectures demonstrate distinct categorical strengths: sequence-based models particularly excel in "Input Validation", whereas graph-based models show broader competency across multiple categories, including "Input Validation" and "API Abuse". (3) The categorical performance patterns align with the individual vulnerability analysis, reinforcing the complementary nature of different model architectures.

**Finding 2:** (1) Different models demonstrate distinct advantages in detecting specific vulnerability types, reflecting their complementary detection capabilities. Specifically, sequence-based models exhibit particular proficiency in "Input Validation" vulnerabilities, while graph-based models demonstrate broader competency across multiple categories (e.g., "Input Validation" and "API Abuse"). (2) Overall, sequence-based models, particularly SVulD, achieve superior performance and show promising potential for practical deployment when detecting the most dangerous vulnerabilities in real-world scenarios.

**•[RQ-3]: Are large language models capable of detecting vulnerabilities under the prompt engineering?**

**Objective.** Large Language Models (LLMs) (Brown et al., 2020) have demonstrated remarkable capabilities across diverse software engineering tasks, benefiting from advances in Natural Language Processing that enable training with billions of parameters and training samples. LLMs can be easily used for a downstream task by being prompted (Liu et al., 2023; Yin, 2024), with previous studies (Liu et al., 2021; Lu et al., 2021; Yin et al., 2024b) showing that LLM performance can vary significantly based on prompting approaches. While vulnerability detection represents a critical and complex reasoning task, no comprehensive study has systematically investigated how different prompting strategies affect LLM performance in this domain. Therefore, we aim to conduct a thorough investigation of LLM capabilities for vulnerability detection under various prompt engineering approaches, providing insights into the most effective strategies for application.

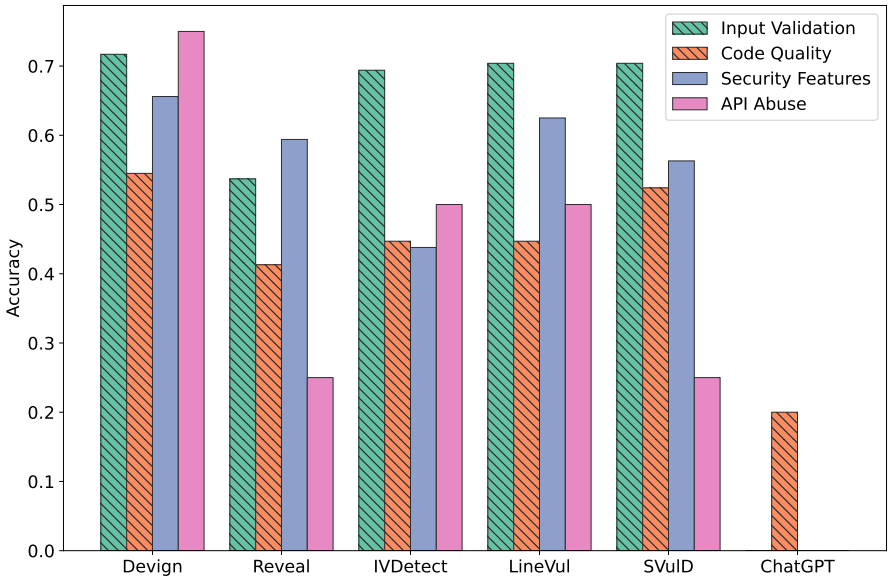



**Fig. 1:** Performance on Vulnerability Type

**Supplementary Setting.** We evaluate the state-of-the-art LLMs, including CodeLlama-34B (Roziere et al., 2023), DeepSeek-Coder-33B (AI, 2023), and ChatGPT (OpenAI, 2022).

To systematically investigate the impact of prompt engineering on vulnerability detection, we adopt three complementary prompting strategies, each chosen based on established theoretical foundations and empirical evidence: (1) **Zero-Shot prompting** serves as a baseline to evaluate the inherent vulnerability detection capabilities of LLMs without demonstrations, allowing us to assess their fundamental understanding of vulnerability patterns. (2) **In-Context Learning (ICL)** is included based on Brown et al.'s (Brown et al., 2020) seminal findings that providing few-shot examples significantly enhances LLM performance on downstream tasks by enabling implicit learning from demonstrations. (3) **Chain-of-Thought (CoT)** is adopted following Wei et al.'s (Wei et al., 2022) evidence that explicit reasoning processes substantially improve LLM performance on complex reasoning tasks. Vulnerability detection, with its demand for multi-step security analysis, aligns well with this approach, making CoT a valuable strategy. 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).

For Few-Shot settings (i.e., Few-Shot ICL and Few-Shot CoT), the selection of appropriate demonstrations critically impacts performance (Min et al., 2022). We design five distinct selection strategies to systematically explore different

factors that potentially influence LLM vulnerability detection performance. The details of each strategy and its theoretical foundations are 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 limitations are reached. These CWEs are selected from the Top-25 Most Dangerous Software Weaknesses.
- **Random Selection.** We randomly sample demonstrations from the training set to serve as a baseline, providing a fair comparison point for other strategies.
- **Random$_{repo}$ Selection**. We randomly select demonstrations from the training set, and these demonstrations are from the same projects that the target function belongs to. This strategy tests the hypothesis that domain-specific or project-specific examples enhance LLM performance by providing contextually relevant coding patterns and vulnerability manifestations.
- **Diversity-based Selection.** We adopt a pre-trained model (i.e., CodeBERT (Feng et al., 2020b)) to embed all the functions from the training set and then use the 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. This approach is grounded in the principle that diverse examples provide broader coverage of the vulnerability detection patterns, potentially improving generalization.
- **Semantic-based Selection.** We utilize CodeBERT embeddings to select demonstrations with the highest cosine similarity to the target function. This strategy is based on the hypothesis that semantically similar examples facilitate better pattern learning for vulnerability detection.

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 the 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

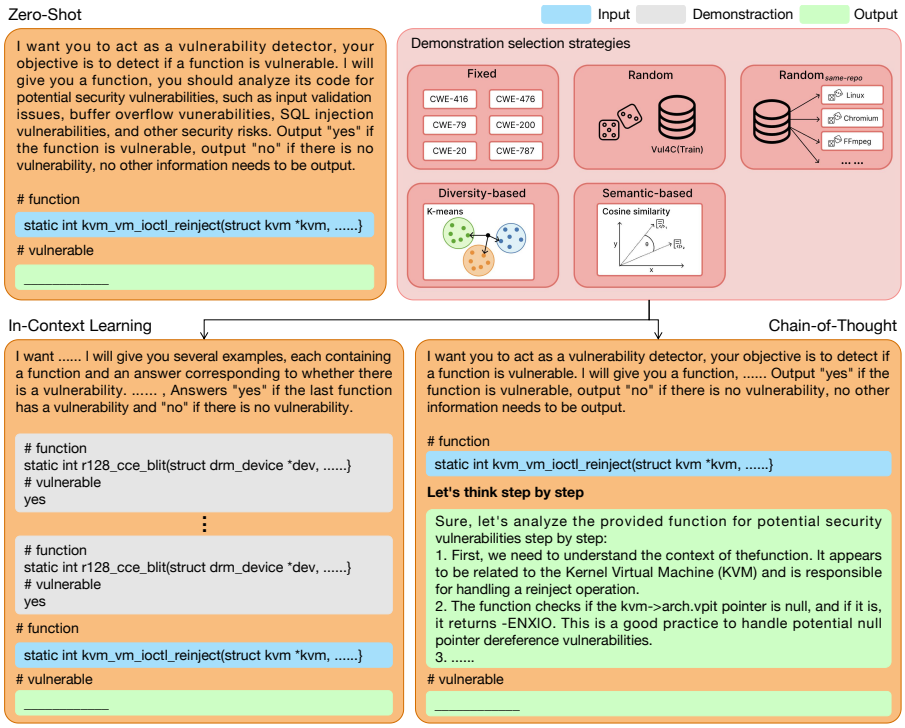



**Fig. 2:** Vulnerability detection prompt templates used in our study

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 reasoning process. These demonstrations and reasoning 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 make 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 settings. (3) When prompted with in-context

**Table 7:** The prompt design for ChatGPT when detecting vulnerability

| ZoSt | ICL | CoT | Example Selection Strategy | | | | | Accuracy | Recall | Precision | F1-score |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Fixed | Rdm | $\text{Rdm}_{repo}$ | 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-score |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Fixed | Rdm | $\text{Rdm}_{repo}$ | 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 |

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

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

| ZoSt | ICL | CoT | Example Selection Strategy | | | | | Accuracy | Recall | Precision | F1-score |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Fixed | Rdm | $\text{Rdm}_{repo}$ | 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** |

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

learning and the $\text{Random}_{repo}$ strategy, ChatGPT performs well, achieving a *Precision* of 0.057, an *F1-score* of 0.078, and a *Recall* of 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 clean ones. 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., 2.9% vulnerability in our dataset), CodeLlama prompted with in-context learning as well as *Diversity-based* strategy is the best setting.

**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 $\text{Random}_{repo}$ selection strategy performs the best in terms of *Precision* and *F1-score*.

### 4.2 D2: Interpretation of Learning-based Models for Vulnerability Detection

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

**Objective.** Effective vulnerability detection models should provide interpretable insights into their decision-making processes, enabling developers to understand which code patterns trigger vulnerability predictions. While various interpretability techniques have been applied to vulnerability detection models (Ying et al., 2019; Shrikumar et al., 2019; Zou et al., 2021; Hu et al., 2023; Li et al., 2021a; Fu and Tantithamthavorn, 2022), comprehensive studies examining the interpretability of different learning-based approaches remain limited. Furthermore, there is a lack of systematic investigation into how different model architectures focus on various statement types during vulnerability detection. Therefore, it is meaningful to investigate whether different deep learning models make decisions based on specific types of statements and help the model be better understood. For instance, the model might pay more attention to "if" statements when detecting input validation vulnerabilities. Additionally, different types of learning-based models (i.e., graph-based and sequence-based) may focus on varying types of information, and figuring out the differences in code features concerned by models can help to improve their abilities.

**Supplementary Setting.** 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 Devign and IVDetect, 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 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 and SVulD), we use the attention layer to get each token's score since they are Transformer-based models (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 15 types of statements (shown in Table 10) within the functions. For the top 10 lines, we then calculate the proportion of each statement type's occurrences relative to its total count.

**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 |

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 sequence-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 proportion of the two statement types exceeds 40% among the studied models, which suggests that both operations will introduce unstable factors to functionality and are prone to introducing vulnerabilities. For example, Fig. 4 shows a function from the Linux project that aims to parse the channel attribute in the WIFI configuration. The

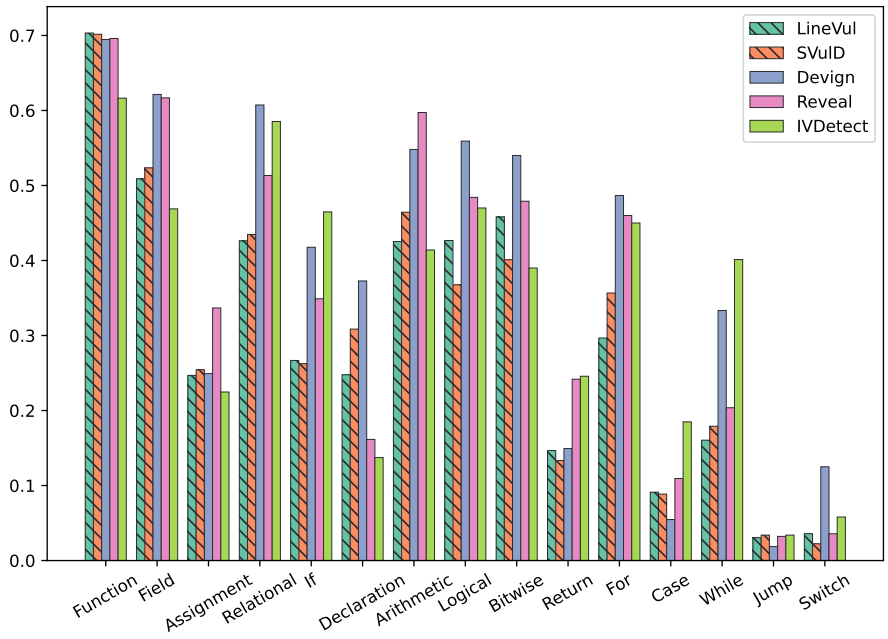



**Fig. 3:** Number of occurrences for each statement type in the Top 10 most probable vulnerability lines (diagonal shadow indicates sequence-based models)

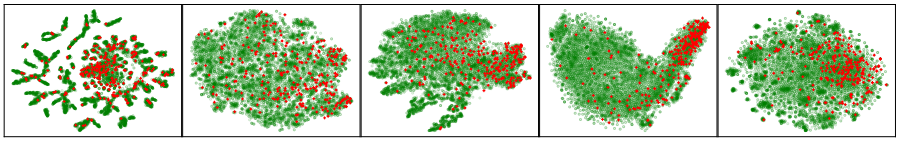
```
1  static inline void wilc_wfi_cfg_parse_ch_attr(u8 ......
2  {
     ......
13   while (index + sizeof(*e) <= len) {
14     e = (struct wilc_attr_entry *)&buf[index];
15     if (e->attr_type == IEEE80211_P2P_ATTR_CHANNEL_LIST)
16       ch_list_idx = index;
17     else if (e->attr_type == IEEE80211_P2P_ATTR_OPER_CHANNEL)
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)

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 "return", "case", "while", "jump", and "switch" statement types. We conduct 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 them are built upon CodeBERT (Feng et al., 2020a) and variants (Guo et al., 2022). For example, for each type of statement, the two models 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 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.

**Finding 4:** (1) Both graph-based and sequence-based models technically focus on two types of statements: Function Calls and Field Expressions, which may involve vulnerable or incredible operations on functionality. (2) The existing learning-based models still have a limited ability to distinguish vulnerable functions from non-vulnerable functions. Sequence-based models perform better than the graph-based models.



(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 ◯)

### 4.3 D3: Robustness of Learning-based Models for Vulnerability Detection

• **[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 robustness of the studied models by evaluating their generalizability to slightly modified but semantically equivalent input.

**Supplementary Setting.** We adopt a comprehensive performance metric F1-score to analyze the performance differences. To evaluate model robustness, we design three categories of experiments. The first category tests whether models are sensitive to minor input variations through semantic-preserving transformations. The second category examines model sensitivity to different training/testing data splits using 10-fold cross-validation. The third category investigates whether models can quickly learn a CWE pattern from a single CVE instance and generalize to detect other vulnerabilities of the same type.

First, 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 4.1 on the four variant test sets.

Second, we perform 10-fold cross-validation to observe the impact of different training/testing data splits.

Third, to investigate whether models can learn from a single CVE pattern within a CWE and quickly generalize to detect other vulnerabilities of the same type, we conduct a new experiment. For each CWE type in the training set, we randomly select one vulnerable code sample and combine it with non-vulnerable code samples to form a new training set. We also maintain the original distribution of vulnerabilities (i.e., vul:non-vul ratio) found in the full training set. We then evaluate model performance by training on this sampled training set to assess the model's capability to generalize from minimal vulnerable examples. This experimental design simulates real-world scenarios where security teams need to quickly adapt their detection systems upon discovering the first instance of a new vulnerability pattern, making it particularly relevant for practical deployment in dynamic environments.

**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' robustness than other types of transformations.

To further evaluate model robustness across different data distributions, we conduct 10-fold cross-validation experiments on all studied models. Table 13 presents the F1-score performance of each model across the 10 different folds, along with the mean performance and standard deviation. The results reveal several key observations regarding model robustness: (1) **Sequence-based models demonstrate superior robustness compared to graph-based models.** The standard deviations for sequence-based models range from 0.005 to 0.007, while graph-based models consistently exhibit a higher standard

**Table 12:** Impact of semantically-equivalent variants and sampled training set

| Types | Models | Original | Remove All Comments | Insert Comments | Insert Irrelevant Code | Rename All Identifiers | Sampled Training Set |
|---|---|---|---|---|---|---|---|
| **Graph Based** | Devign | 0.122 | 0.075 (38.5%↓) | 0.075 (38.5%↓) | 0.073 (40.2%↓) | 0.075 (38.5%↓) | 0.057 (53.3%↓) |
| | Reveal | 0.125 | 0.099 (20.8%↓) | 0.093 (25.6%↓) | 0.094 (24.8%↓) | 0.097 (22.4%↓) | 0.064 (48.8%↓) |
| | IVDetect | 0.141 | 0.051 (63.8%↓) | 0.052 (63.1%↓) | 0.055 (61.0%↓) | 0.025 (82.3%↓) | 0.066 (53.2%↓) |
| **Sequence Based** | LineVul | 0.195 | 0.196 (0.5%↑) | 0.186 (4.6%↓) | 0.195 (-) | 0.183 (6.2%↓) | 0.126 (35.4%↓) |
| | SVulD | 0.172 | 0.174 (1.2%↑) | 0.163 (5.2%↓) | 0.179 (4.1%↑) | 0.186 (8.1%↑) | 0.118 (31.4%↓) |
| | CodeLlama | 0.158 | 0.155 (1.9%↓) | 0.153 (3.2%↓) | 0.155 (1.9%↓) | 0.146 (7.6%↓) | 0.092 (41.8%↓) |
| | DeepSeek-Coder | 0.167 | 0.165 (1.2%↓) | 0.160 (4.2%↓) | 0.162 (3.0%↓) | 0.151 (9.6%↓) | 0.101 (39.5%↓) |
| | Magicoder | 0.151 | 0.147 (2.6%↓) | 0.143 (5.3%↓) | 0.145 (4.0%↓) | 0.141 (6.6%↓) | 0.095 (37.1%↓) |
| | 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.

deviation of 0.010. (2) **Among sequence-based models, LineVul and SVulD achieve the highest robustness** with the lowest standard deviation (0.005), suggesting their robust performance across different data splits. (3) **Graph-based models show consistent but higher variability** with all three models (Devign, Reveal, and IVDetect) exhibiting identical standard deviations of 0.010, indicating similar levels of instability across different data distributions.

**Table 13:** Model performance robustness across 10-fold cross-validation

| Fold | Graph Based | | | Sequence Based | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Devign | Reveal | IVDetect | LineVul | SVulD | CodeLlama | DeepSeek-Coder | Magicoder | ChatGPT* |
| 1 | 0.101 | 0.125 | 0.139 | 0.193 | 0.177 | 0.144 | 0.163 | 0.161 | 0.072 |
| 2 | 0.115 | 0.121 | 0.120 | 0.196 | 0.176 | 0.162 | 0.177 | 0.155 | 0.078 |
| 3 | 0.137 | 0.130 | 0.126 | 0.189 | 0.180 | 0.160 | 0.159 | 0.142 | 0.069 |
| 4 | 0.119 | 0.127 | 0.135 | 0.183 | 0.172 | 0.161 | 0.165 | 0.152 | 0.079 |
| 5 | 0.126 | 0.118 | 0.133 | 0.185 | 0.162 | 0.153 | 0.168 | 0.153 | 0.089 |
| 6 | 0.125 | 0.126 | 0.131 | 0.193 | 0.172 | 0.157 | 0.167 | 0.149 | 0.077 |
| 7 | 0.122 | 0.102 | 0.148 | 0.195 | 0.176 | 0.159 | 0.171 | 0.148 | 0.078 |
| 8 | 0.129 | 0.142 | 0.151 | 0.188 | 0.174 | 0.168 | 0.175 | 0.163 | 0.062 |
| 9 | 0.108 | 0.120 | 0.132 | 0.184 | 0.178 | 0.158 | 0.164 | 0.147 | 0.071 |
| 10 | 0.130 | 0.128 | 0.120 | 0.179 | 0.182 | 0.152 | 0.177 | 0.142 | 0.075 |
| Mean | 0.121 | 0.124 | 0.134 | 0.189 | 0.175 | 0.157 | 0.169 | 0.151 | 0.075 |
| Std Dev | 0.010 | 0.010 | 0.010 | 0.005 | 0.005 | 0.006 | 0.006 | 0.007 | 0.007 |

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

From the results in the last column of Table 12, we find that when trained on a minimal training set (one vulnerable sample per CWE), all models experience substantial performance drops ranging from 31.4% to 53.3%. Graph-based models show more severe degradation (i.e., 48.8% to 53.3%) compared to sequence-based models (i.e., 31.4% to 41.8%). This indicates that current vulnerability detection models heavily rely on abundant training samples and struggle to generalize from minimal vulnerability patterns, highlighting a significant limitation for real-world deployment scenarios where new vulnerability types are initially represented by very few samples.

**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.

### 4.4 D4: Ease of Deployment of Learning-based Models for Vulnerability Detection

• **[RQ-6]: What types of efforts should be paid before using learning-based models? 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.

**Supplementary Setting.** 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.

**Table 14:** Ease of Deployment of Learning-based Models

| Models | Input Requirements | | | Model Features | | | | |
|---|---|---|---|---|---|---|---|---|
| | Program Integrity | Compilation | Input Size | Training/ 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 |

**Results.** We summarize the ease of deployment of the models in Table 14. From the results, we obtain the following conclusions: (1) Graph-based models require complete input programs since their inputs must be successfully parsed into 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 ($\leq$16K tokens), while sequence-based models LineVul and SVulD are limited to a small input size ($\leq$ 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 training or 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 the 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.

Our findings provide actionable guidance for diverse real-world deployment contexts: (1) **Enterprise Deployment Decision-making.** Software companies need to select appropriate models based on their infrastructure constraints, security requirements, and input characteristics. Our analysis helps organizations make informed decisions by clearly outlining hardware requirements (1GB for graph-based vs. 6GB+ for sequence-based vs. 72GB+ for LLMs), privacy implications (local deployment vs. API-based solutions), and input limitations. Organizations with limited computational resources can use our hardware requirement analysis to determine feasible solutions, while those with privacy-critical codebases can prioritize local deployable models over cloud-based APIs. (2) **Tool Integration.** IDE plugin developers and static analysis tool vendors require practical guidance on which models can handle incomplete code snippets, partial functions, or real-time analysis scenarios. Our findings that sequence-based models can process incomplete programs while graph-based models require program integrity directly impact tool design decisions.

**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 training or 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.

### 4.5 D5: Economy of Learning-based Models for Vulnerability Detection

**•[RQ-7]: What are the costs caused by learning-based 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.

**Supplementary Setting.** Given the significant performance difference between LLMs in fine-tuning and prompt engineering, this RQ focuses exclusively on learning-based models within a training/fine-tuning setting. 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 training 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. For the learning-based models, we calculate the cost of going from preprocess data to inference the 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 15:** 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 |

**Results.** The results are summarized in Table 15. 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 structure 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 does not adopt an early stopping mechanism. (3) Among all the models, SVulD is the most economical option with a cost of only 1.78$, 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, SVulD is the most economical solution.

## 5 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-0125). 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.

A significant threat concerns the limitations of our economic analysis in RQ-7, which provides a simplified baseline comparison that may not fully reflect real-world deployment scenarios. Our cost analysis focuses primarily on computational costs during active inference periods and does not account for several critical factors in production environments: (1) **Continuous service costs** - cloud instances must remain online 24/7 regardless of request volume, incurring baseline infrastructure costs; (2) **Operational maintenance costs** - system monitoring, updates, security patches, and technical support; (3) **Data hosting and transmission costs** - model storage, data transfer, and backup expenses. Additionally, realistic usage patterns derived from actual software development practices (e.g., commit frequencies in popular repositories) could significantly impact cost-effectiveness calculations compared to our one-time inference analysis.

Despite these limitations, our simplified analysis serves as a preliminary benchmark for comparing relative computational costs across different model architectures. While not capturing full deployment economics, it provides preliminary insights for researchers and practitioners to understand the computational trade-offs between different vulnerability detection approaches.

## 6 Conclusion

This paper aims to comprehensively investigate the capabilities of graph-based and sequence-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 capability*, *model interpretation*, *model robustness*, *ease of model deployment*, 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.

## Declarations

**Funding.** This work was supported by Zhejiang Pioneer (Jianbing) Project (2025C01198(SD2)), the National Natural Science Foundation of China (Grant No.62202419), the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), Zhejiang Provincial Natural Science Foundation of China (No. LY24F020008), the Ningbo Natural Science Foundation (No. 2022J184), the Key Research and Development Program of Zhejiang Province (No.2021C01105), and the State Street Zhejiang University Technology Center.

**Ethical approval.** All procedures performed in studies involving human participants were in accordance with the ethical standards of the institutional and/or national research committee.

**Informed consent.** Written informed consent was obtained from all authors for the publication of this paper.

**Author Contributions.** Chao Ni is the corresponding author. Xin Yin and Liyu Shen co-designed the experiment and wrote the paper. Shaohua Wang participated in the idea proposal stage of the paper.

**Data Availability Statement.** For code and data availability, we release our reproduction package: `https://github.com/vinci-grape/`

Learning-based-Models-for-VD, including the datasets and the source code, to facilitate other researchers and practitioners to repeat our work and verify their studies.

**Conflict of Interest.** Beyond this, the authors have no conflicts of interest to declare that are relevant to the content of this article.

**Clinical Trial Number.** Not applicable.

## 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) 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

Ding Y, Fu Y, Ibrahim O, Sitawarin C, Chen X, Alomair B, Wagner D, Ray B, Chen Y (2024) Vulnerability detection with code language models: How far are we? arXiv preprint arXiv:240318624

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

Hu Y, Wang S, Li W, Peng J, Wu Y, Zou D, Jin H (2023) Interpreters for gnn-based vulnerability detection: Are we there yet? In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp 1407–1419

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

Mazuera-Rozo A, Mojica-Hanke A, Linares-Vásquez M, Bavota G (2021) Shallow or deep? an empirical study on detecting vulnerabilities using deep learning. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), IEEE, pp 276–287

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)

Ni C, Yin X, Li X, Xu X, Yu Z (2025) Abundant modalities offer more nutrients: Multi-modal-based function-level vulnerability detection. ACM Transactions on Software Engineering and Methodology

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 Z, Bourgeois D, You J, Zitnik M, Leskovec J (2019) Gnnexplainer: Generating explanations for graph neural networks. Advances in neural information processing systems 32

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

Zou D, Zhu Y, Xu S, Li Z, Jin H, Ye H (2021) Interpreting deep learning-based vulnerability detector predictions based on heuristic searching. ACM Transactions on Software Engineering and Methodology (TOSEM) 30(2):1–31