On the Effectiveness of Instruction-Tuning Local LLMs for Identifying Software Vulnerabilities*

Sangryu Park, Gihyuk Ko, and Homook Cho †

KAIST Cyber Security Research Center, Daejeon, Republic of Korea
{sangryupark, gihyuk.ko, chmook79}@kaist.ac.kr

Abstract

Large Language Models (LLMs) show significant promise in automating software vulnerability analysis, a critical task given the impact of security failure of modern software systems. However, current approaches in using LLMs to automate vulnerability analysis mostly rely on using online API-based LLM services, requiring the user to disclose the source code in development. Moreover, they predominantly frame the task as a binary classification (vulnerable or not vulnerable), limiting potential practical utility. This paper addresses these limitations by reformulating the problem as Software Vulnerability Identification (SVI), where LLMs are asked to output the type of weakness in Common Weakness Enumeration (CWE) IDs rather than simply indicating the presence or absence of a vulnerability. We also tackle the reliance on large, API-based LLMs by demonstrating that instruction-tuning smaller, locally deployable LLMs can achieve superior identification performance. In our analysis, instruct-tuning a local LLM showed better overall performance and cost trade-off than online API-based LLMs. Our findings indicate that instruct-tuned local models represent a more effective, secure, and practical approach for leveraging LLMs in real-world vulnerability management workflows.

Keywords: Large Language Models (LLMs), Software Vulnerability, Vulnerability Detection, Vulnerability Identification, Local LLMs

1 Introduction

The growing reliance on software across all areas of society has driven the development of increasingly complex and high-performing software systems. However, the risk of security failure grows alongside this advancement as larger and more complex software tend to harbor more security vulnerabilities [24]. As a result, they often become attractive targets for malicious attackers seeking to exploit them [37]. Reflecting this trend, a number of published CVE (Common Vulnerabilities and Exposures) records has more than tripled in the last decade, increasing from 59,487 (2005–2014) to 194,049 (2015–2024) records [1]. Yet, to this day, effectively detecting and mitigating security vulnerabilities in software systems remains a pervasive challenge.

To combat this, a numerous Static Application Security Testing (SAST) [19, 12, 28, 21, 7] methods have been suggested over time. While they provide certain guarantee, they often suffer from inherent limitations such as high false-positive rates, context blindness, late-stage application, and lack of code-level precision [7, 20, 15]. While Machine Learning (ML) and Deep Learning (DL) have shown promise in overcoming some of these hurdles, the advent of

*Proceedings of the 9th International Conference on Mobile Internet Security (MobiSec'25), Article No. 9, December 16-18, 2025, Sapporo , Japan. © The copyright of this paper remains with the author(s).

†Corresponding Authors

Large Language Models (LLMs) represents a potentially transformative shift in automated vulnerability analysis due to their advanced code understanding capabilities [33, 3, 14].

Despite the potential of LLMs, current research on automating source code vulnerability analysis with LLMs possess a few limitations. First, many proposed works rely on online API-based LLM services such as GPT-4o and Gemini [35, 42, 43, 11], raising concerns about privacy and intellectual property [22]. Additionally, many works predominantly frame software vulnerability detection as a simple binary classification problem – determining if an isolated code snippet is simply “vulnerable” or “not vulnerable” [33, 14]. This simplification, often driven by benchmark dataset structures and ease of evaluation, can cause limited actionability for remediation.

This paper addresses these critical gaps through following contributions. Firstly, we focus on and reformulate the problem of Software Vulnerability Identification (SVI), where we prompt LLMs to predict not only whether a given piece of source code is vulnerable, but also to identify the specific type of vulnerability. Secondly, we challenge the reliance on larger, online API-based LLMs which entail significant cost and privacy concerns. We investigate and demonstrate the effectiveness of instruct-tuning smaller, locally deployable LLMs, showing they can achieve superior performance on this specialized task. Lastly, we provide analysis Common Weakness Enumeration (CWE) hierarchy as a framework for a structured analysis of our model's identification capabilities across different weakness categories and abstraction levels, moving beyond simplistic aggregate metrics. By addressing these gaps, this work aims to advance LLM-based vulnerability analysis towards more practical, secure, and effective solutions.

The remainder of this paper is organized as follows. Sections 2 and 3 provide background for our work by reviewing related work and outlining the problem setting, respectively. Section 4 presents our approach to instruction-tuning smaller, locally deployable LLMs for Software Vulnerability Identification (SVI). In Section 5, we detail the datasets and models used for evaluation. Sections 6 and 7 report the experimental results and offer corresponding discussions.

2 Related Works

Over the recent years, many works have been suggested to effectively analyze software vulnerability using Large Language Models (LLMs). In this section, we review previously suggested LLM-based vulnerability detection methods from the perspective of three areas: discriminative model-based, generative model-based, and LLM agent-based.

2.1 Discriminative Model-based Vulnerability Detection

Early works in LLM-based vulnerability detection primarily used discriminative models such as BERT, which are well-suited for classification and detection tasks. For instance, Kim et al. [18] proposed VulDeBERT, which detects code vulnerabilty by slicing the target program into code gadgets and embedding them using BERT-based model. Since it relied on system calls to detect vulnerabilities, VulDeBERT's detection scope is limited. Similarly, Fu et al. [10] introduced LineVul, a CodeBERT-based line-level vulnerability detection method. LineVul uses CodeBERT to generate vector representations of code lines and checked if each line had any vulnerabilities. However, due to parametrical constraints of BERT, LineVul is limited to handling inputs of up to 512 tokens, making it unsuitable for analyzing longer code samples.

2.2 Generative Model-based Vulnerability Detection

Generative large language models were also actively utilized to detect code vulnerabilities. For instance, Zhou et al. [44] examined vulnerability detection performance of generative language models such as GPT-3.5 and GPT-4, alongside with CodeBERT. For vulnerability detection, GPT-3.5 and GPT-4 outperformed state-of-the-art vulnerability detection model based on CodeBERT. However, Zhou et al. pointed out that using API-based online LLM services such as GPT-like models can cause data privacy and security issues since thet require sending code to external servers.

To address the limitations of single-task models, Du et al. [6] introduced VulLLM that treats vulnerability detection as a multi-task problem. Unlike other works, VulLLM was capable of localizing the potentially vulnerable lines, detecting vulnerabilities and generating an explanation for it. However, VulLLM had an overfitting problem where the accuracy of VulLLM significantly drops when detecting vulnerability data from out of distribution.

2.3 LLM Agent-based Vulnerability Detection

Currently many of LLM works are based on LLM agents and also a number of vulnerability detection research has been published. Purba et al. [5] presents a multi-component LLM agent system for vulnerability detection, where it uses checklist-guided LLM agents to analyze source code. On the other hand, Guo et al. [13] proposed RepoAudit, where autonomous LLM agents audit repository-level code to find vulnerabilities. RepoAudit introduces dual validation to mitigate hallucinations and employs demand-driven graph traversal to enhance scalability, achieving an average analysis time of just 0.44 hours for 251,000 code files. Additionally, RepoAudit claims that it can utilize LLM's implicit path discrimination without explicit enumeration of sensitive paths.

While effective in certain scenarios, agent-based approaches can also face notable limitations. Notably, Yarra [41] points out that the performance of LLM agent-based vulnerabilty detection is often dependent on the underlying LLM, which may result in a high false negative rates. Furthermore, Yarra [41] claims that agent-based methods can be more challenging to design, deploy, and maintain, as the architectures typically requires multiple LLM inferences per vulnerability. This can introduce significant computational overhead in terms of both cost and time. In particular, the total processing time and cost tend to increase roughly in proportion to the number of agents involved.

3 Problem Setting

In this section, we outline the problem setting under investigation, including the assumptions, objectives, and settings relevant to our study.

Local Analysis of Software Vulnerabilities In this work, we consider a situation where a user wishes to analyze software vulnerabilities within a closed environment – that is, without disclosing the target source code. We argue that this is a highly plausible scenario, as organizations – particularly those developing innovative or sensitive technologies – may prefer to keep their products confidential until a full public launch. Since it is a well-known fact that approximately 70% of software vulnerabilities stem from the defects in the development phase [3], they have much incentives to deploy their own local LLM instead of having to disclose their source codes online.

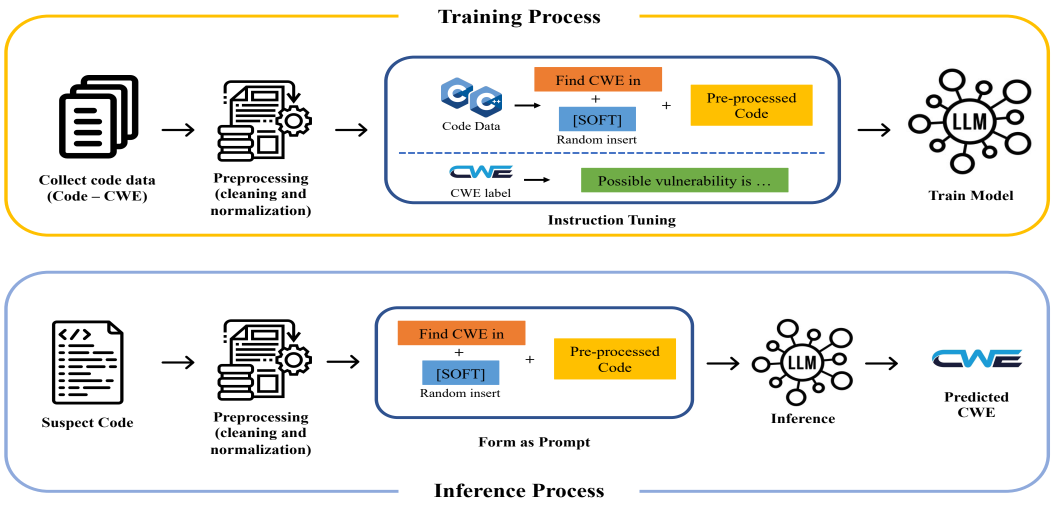

Figure 1: Fine-tuning LLMs for Software Vulnerability Identification

Software Vulnerability Identification (SVI) In this work, we aim to address the problem of *Software Vulnerability Identification (SVI)* for source code. Software Vulnerability Identification (SVI) is similar to Software Vulnerability Detection (SVD) in that SVD aims to detect *whether or not* a given source code is vulnerable, but it also aims to specify *which type of vulnerabilty* the given source code has. Naturally, while SVD can be formulated as a binary classification task, SVI can be formulated as a multi-class classification task.

Formally, we define the goal of the user as the following: Let $\{s_i\}$ denote the set of source codes the user wishes to check and $\{v_i\}$ be the corresponding vulnerability types (it can be benign). Let C_{LLM} denote a classifier identifying vulnerabilities based on LLM. The user's goal in our setting will be to minimize $\sum_i \mathcal{L}(v_i, \bar{v}_i)$, where $\bar{v}_i = C_{LLM}(s_i)$ and $\mathcal{L}(\cdot)$ is an appropriate loss function.

Common Weakness Enumeration (CWE) Common Weakness Enumeration (CWE) [2] is a structured list and classification system for vulnerabilities that includes over 600 types of vulnerabilities and provides taxonomy. In our problem setting, we assume that the objective of the model is to identify CWEs. While Common Vulnerabilities and Exposures (CVEs) are a more specific and practical list of weaknesses list than CWEs, it is difficult to efficiently learn CVEs as they are often too specific to the hardware and software configurations.

4 Our Method

In this section, we detail our method of using LLMs for software vulnerability identification. In our approach, we adopt an instruct-tuning [40] method, where each data in a labeled dataset is first formatted in a predefined textual format, and then used to fine-tune the backbone LLM. We argue that since most decoder-based LLMs (which most modern generative language models are based on) are trained for *generative* tasks rather than *discriminative* tasks, our prompt-tuning method can be efficiently applied to enhance vulnerability identification capabilities of

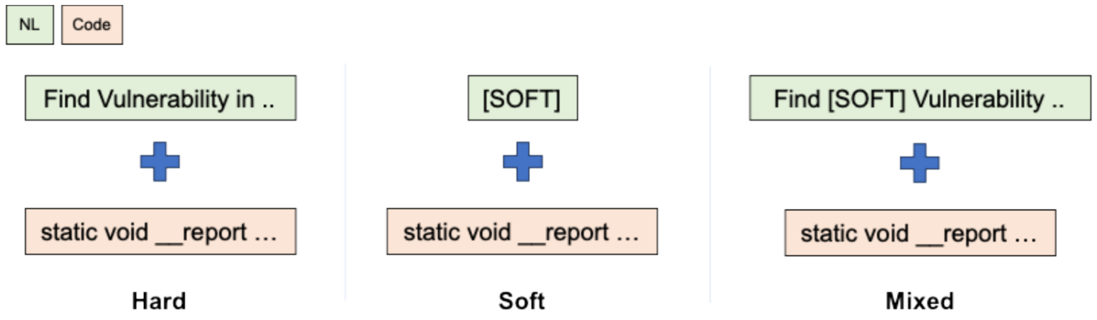

Figure 2: Different prompting styles using [SOFT] token

LLMs.

An overview of our approach is presented in Fig. 1. Given a dataset of [Code–CWE] pairs, we follow standard procedure to prepare code with textual prompts for instruction tuning. We provide additional detail on the collected dataset in Section 5.

4.1 Preprocessing codes

We first remove unnecessary or noisy texts in the source code to prevent them from disturbing training. For instance, special characters such as newline (`\n`) and tab (`\t`) are removed. Moreover, we normalize indentation by changing multiple tabs or spaces into a single space per indent level. Lastly, as the comments in the source code can be hinting towards the CWE IDs, they have been removed.

4.2 Prompting

Next, we combine preprocessed codes with textual queries to form prompts. Following [16], we consider three different styles of prompting: `hard`, `soft`, and `mixed prompting`. While textual query is used as-is in `hard prompting`, the query is replaced with a specially designated *[SOFT]* token in `soft prompting`. In `mixed prompting`, *[SOFT]* tokens are randomly inserted in the textual queries. Fig.2 illustrates three different prompting styles.

We supplement each prompt with an expected response, which contains either the description and/or the ID of the corresponding vulnerability (i.e., CWE), or a message denoting that the code is not vulnerable. Note that there could be three variants of the expected response: (1) CWE-ID with description, (2) CWE-ID only, and (3) CWE description only. From our initial experiment, CWE-IDs seem to confuse language models as the CWE IDs themselves are simply numeric, and do not entail meaningful information. Therefore, we decided to use "3) CWE description only" response.

4.3 Instruction Tuning and Post-Processing

Lastly, we instruct-tune the model with prompt-augmented code data. During the instruct-tuning process, the model is trained to extract features from input code prompt and return CWE description as an answer. We apply a standard instruction-tuning which corrects parameters based on how close the current textual output is to the ground truth. Since outputs are purely textual, we used BLEU score [29] to measure the discrepancies.

Table 1: Composition of the collected dataset.

CWE ID	**Description**	**Rank**	**Counts**
CWE-787	Out of Bounds Write	2	31,692
CWE-125	Out of Bounds Read	6	23,161
CWE-416	Use After Free	8	17,894
CWE-20	Improper Input Validation	12	18,739
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	17	10,890
CWE-119	Improper Restriction of Operation within Bounds of Memory Buffer	20	21,937
CWE-476	Null Pointer Dereference	21	15,121
CWE-190	Integer Overflow or Wraparound	23	9,384
CWE-703	Improper Check or Handling of Exceptional Conditions	-	19,910
Benign			18,823
Total			**187,551**

During the inference phase, a well-trained model may generate a CWE description that differs slightly from the ground truth. To address this, we implemented a post-processing mechanism that selects the closest matching CWE description from a predefined set of possible descriptions. Our post-processing first selects the most similar description based on the matching word counts. If there exist multiple candidates with the same number of matching words, we select the candidate with the highest BLEU score [29].

5 Data Collection and Model

Collecting sufficient, high-quality data samples and selecting a model architecture well-suited to the task is a prerequisite for building a successful AI system. This is especially true for Software Vulnerability Identification, as imbalances between vulnerable and non-vulnerable code samples, or wrong choice of the model architecture can significantly affect model performances. In this section, we describe the process of constructing our dataset and explain our rationale for choosing the base models.

5.1 Dataset

We design our model to identify vulnerabilities in code snippets written in C/C++, as most datasets used in automatic vulnerability detection (or identification) consist predominantly of C/C++ code [33]. In our evaluation, we collected vulnerable code data from BigVul [8], DiverseVul [4], and SVEN [17]. To balance the dataset with benign code samples, we collected code snippets from GNU Coreutils.

In selecting the target weaknesses (i.e., CWEs) for vulnerable code samples, we consider two key factors: (1) sufficiency of the code samples for each CWE type, and (2) the importance of the weakness in real-world scenarios. To ensure effective training of LLMs, we prioritize CWEs with approximately 10,000 or more code samples. Additionally, to reflect real-world threat patterns, we prioritize those CWEs that appear in the CWE *Top 25 Most Dangerous Software Weaknesses* list [38].

Table 1 shows the composition of our dataset. Our dataset consists of 187,551 code snippets with 10 different labels: 9 corresponding to different CWE IDs and 1 representing benign code. We record the CWE IDs alongside the description and rank (according to [38]) of each weakness. For consistent evaluation, we randomly select 500 code snippets for each label and form a test dataset consisting of total 5,000 code snippets. The remainder (total of 182,551 code snippets) were used to train the backbone LLM model.

5.2 Backbone Model

The architecture of Large Language Models (LLMs) varies by model, with each offering distinct advantages depending on the underlying task. In the context of Software Vulnerability Identification, encoder-based models such as CodeBERT [9] are particularly effective due to their ability to learn rich semantic features from textual input. These models can be optimized for comprehension tasks, enabling a deeper understanding of source code and facilitating accurate vulnerability identification.

In contrast, decoder-based models such as CodeLlama [31] are typically specialized for generative tasks, including code generation and translation. Although they are not primarily designed for feature extraction, they can be trained with remembering substantial background knowledge. Moreover, thanks to the use of prompting techniques, decoder-based models can be tuned to perform numerous tasks beyond generation, including Software Vulnerability Identification.

In our experiments, we select CodeT5 (`codet5-large`) [39] as our primary target of evaluation. We argue that since CodeT5 is based on the encoder-decoder model T5 [30], it can leverage the strengths of both encoder- and decoder-based models. Specifically, the encoder is responsible for extracting meaningful representations from the input code, while the decoder utilizes this information, along with its pretrained knowledge, to generate task-specific outputs.

To compare our method's efficacy, we also test on 5 different LLMs: CodeBERT (`codebert-base`) [9], GPT-3.5 (`gpt-3.5-turbo`) [26], GPT-4 (`gpt-4-0613`) [27], CodeLlama (`codellama-7b-instruct`) [31], and Llama 3 (`llama3.1-8b-instruct`) [36]. Note that GPT-3.5 and GPT-4 are only accessible via paid API calls, and LLama 3 and CodeLlama are downloadable.

6 Evaluation

In this section, we present the experimental details and results used to assess the effectiveness of our proposed method. Specifically, we design experiments to address the following research questions (RQs):

- **RQ1. (Performance)** How effectively does the instruct-tuned local LLM identify software vulnerabilities compared to online LLMs?
- **RQ2. (Prompting)** Which instruct-tuning method is most effective for identifying software vulnerabilities?
- **RQ3. (Cost-efficiency)** What is the cost efficiency of instruct-tuning a local LLM versus tuning an online LLM?

Table 2: Average performance of vulnerability identification by different LLMs. The results are averaged over five different random seeds to reduce the impact of variance in data sampling.

Model	**CodeT5**	**CodeBERT**	**GPT-3.5**	**GPT-4**	**Code Llama**	**Llama3**
Accuracy	**81.71**	73.36	10.14	11.06	9.62	10.76
Macro-F1	**81.93**	73.53	5.43	6.73	7.79	8.68
FNR	**0.09**	0.11	4.32	13.66	14.69	1.92
FPR	**1.62**	1.81	96.04	87.6	83.88	98.32

Table 3: Label-wise performance of vulnerability identification. Results are based on a single fixed random seed (seed = 42).

CWE ID	**119**	**125**	**190**	**20**	**200**	**416**	**476**	**703**	**787**	**No Vul**
CodeT5	67.80	75.80	75.80	**78.80**	**80.40**	**81.40**	**78.00**	**71.60**	**67.00**	**98.40**
CodeBERT	**74.60**	**76.60**	**80.60**	77.00	71.00	81.00	74.20	67.60	48.80	98.20
GPT-3.5	1.20	4.20	2.20	0.40	0	6.80	66.40	6.80	11.20	3.00
GPT-4	17.00	14.80	5.80	8.60	0.40	10.80	47.60	1.80	1.00	12.40
Code Llama	3.60	12.80	11.00	16.60	6.00	17.00	2.00	16.00	15.20	15.60
Llama3	28.60	14.40	10.20	4.20	1.40	9.40	12.20	15.00	1.40	1.00

Throughout the remainder of this section, we present the results corresponding to each of our research questions.

Experimental setup For all experiments, we used a CodeT5 model (`codet5large`) trained with our dataset, on a AMD EPYC 7543 CPU and 4 NVIDIA-A100 GPUs. We fixed the maximum token length as 1,200, and set batch sizes as 4 for both training and evaluation. We trained the model for maximum 10 epochs with early stopping patience 3 to avoid overfitting.

6.1 RQ1: Vulnerability Identification Performance

The primary objective in automated vulnerability identification is to accurately identify the specific vulnerabilities present in a given code snippet. To address RQ1, we conducted experiments to evaluate the accuracy, Macro-F1 score, False Negative Rate (FNR) and False Positive Rate (FPR) of our model in comparison with several baseline models including CodeBERT, GPT-3.5, GPT-4, CodeLlama, and LLaMA 3, as we denoted in Section 5.2. Note that we fine-tune CodeBERT on the same dataset as CodeT5, but instruction tuning is not applied due to the model's limitations. For the rest models, we query each test sample to obtain textual output, which are post-processed (see Section 4.3) to obtain matching CWE descriptions.

To ensure robustness, we repeated the experiment on 5 differently seeded train-test data splits. In other words, CodeT5 and CodeBERT models were trained on five different training splits, and tested on corresponding test datasets. We record our average performance in Table 2. Note that `accuracy` is an average accuracy of 10 different labels (not equivalent to *vulnerable–*

Table 4: Impact of different prompting methods on the vulnerability identification performances.

Model	`Mixed` **prompt**	**No prompt**	`Hard` **prompt**	`Soft` **prompt**
Accuracy	**77.50**	9.82	72.24	70.80
Macro-F1	**77.58**	8.72	72.53	71.13

Table 5: Impact of textual prompt styles on the vulnerability identification perfomances.

Prompt	**Accuracy**	**Macro-F1**
Simple	**77.50**	**77.58**
C/C++ Specified	74.02	74.22
CodeT5-style	74.94	75.10
Descriptive	73.16	73.42

not vulnerable accuracy). According to Table 2, instruct-tuned CodeT5 model showed the best performance of over 80% accuracy, far exceeding other models. Larger LLMs, believed to have better prediction performance, seemed to suffer in identifying vulnerabilities in the underlying source code.

Table 3 presents per-label accuracy, measured on a single fixed seed. Similar to above, we observed that instruct-tuned CodeT5 shows best or second-best performance in all labels. While CodeBERT shows equally good performance, GPT models seemed to suffer from predicting only one label. From these results, we argue that instruct-tuned local LLMs can be as competent as, or more effective in identifying software vulnerabilities than online, API-based LLMs.

6.2 RQ2: Optimal Prompting

As denoted in Section 4.2, we test various prompt tuning strategies – `hard`, `soft`, and `mixed` prompts – to guide the model behavior. `Hard` prompts provide clear task instructions, helping to constrain the model's focus, while `soft` prompts offer adaptability and can promote more diverse and creative outputs. In this experiment, we evaluated how each prompt method impacts overall accuracy.

Table 4 presents the results. In our experiment, `mixed` prompt method yielded the best performance. We suspect that `mixed` prompting method introduces a degree of variability, allowing the model to treat soft tokens as flexible context or controlled noise, which can enhance its generalization during training.

We also test if different textual prompt styles can affect the detection performance. We test with the following four variants of textual prompts to instruct-tune and query the model:

- **Simple (default)** — `Find CWE in: [CODE]`
- **C/C++ Specified** — `Examine the given C/C++ code snippet and detect vulnerabilities: [CODE]`
- **CodeT5-style** — `Defect: [CODE]`

Table 6: Estimated cost of instruct-tuning LLMs on our vulnerability dataset. The cost has been estimated based on RunPod GPU service [32].

Models	**CodeT5**	**CodeBERT**	**GPT-3.5**	**GPT-4**[g]	**CodeLlama**	**Llama3**
Training	\$431.20[a]	\$3.52[r]	\$9797.70	\$30617.83	\$2304.00[a]	\$2664.00[a]
Inference	\$0.22[r]	\$0.22[r]	\$0.04	\$12.41	\$0.66[r]	\$0.66[r]
Total	\$431.42	\$3.74	\$9797.74	\$30630.24	\$2304.66	\$2664.66

[a] using 4× A100s (80GB ea.)
[r] using 1× RTX3090
[g] GPT-4 does not support fine-tuning service on OpenAI API, therefore calculation is based on GPT-4.1

- **Descriptive** — `Analyze the following code and identify potential security vulnerabilities in: [CODE]`

Table 5 summarizes the results. In our experiments, Simple style of prompting was most effective. We hypothesize that, due to the relatively smaller size of our instruct-tuned local model, descriptive prefixes can disturb the model from correctly reasoning about the code. Note that longer variants (C/C++ Specified and Descriptive) showed lower accuracies in comparison to the shorter variants.

6.3 RQ3: Cost Efficiency

One of the key challenges in training large language models (LLMs) is in the high cost. Models with a large number of parameters and extensive training data require GPUs with substantial VRAM and prolonged training times, which can lead to implosion of expenses. This can further intensify when one has to rely on the paid API-based queries.

To simulate real-world deployment scenarios, we estimated the total cost required to instruct-tune each model on our vulnerability dataset. These estimates are based on actual training durations and the size of the training data. Table 6 summarizes the estimated costs associated with instruct-tuning various LLMs. For both training and inference, we referenced the official pricing of API calls for hosted models, where applicable, or the cost of GPU usage on the RunPod service based on the corresponding GPU (4× A100s for training, 1× RTX3090 for inference).

According to Table 6, locally trained CodeT5 and CodeBERT shows the lowest and second-lowest budget required. This can be expected as the sizes of the local models are much smaller. Considering their high performance, we argue that instruct-tuning local models are more cost-efficient than online models in smaller, domain-specific tasks such as Software Vulnerability Identification.

7 Discussion

In this section, we provide additional discussion on the effectiveness of our approach: instruction-tuning local LLMs for Software Vulnerability Identification. Specifically, we first analyze how CWE's hierarchy is correlated to the error rates. Next, we analyze computational cost of

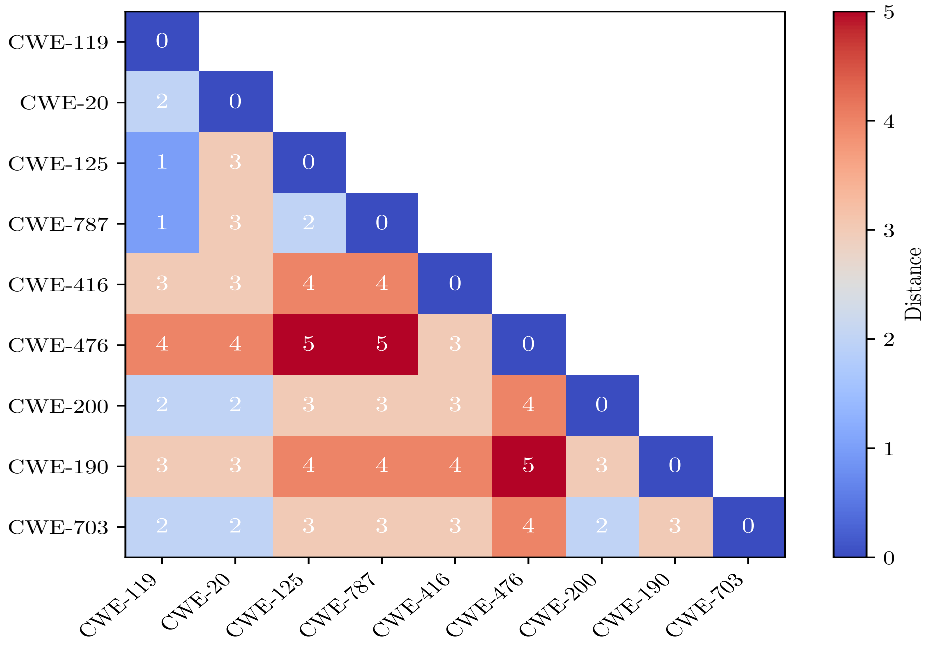

Figure 3: Pairwise CWE Distance Matrix

instruction-tuning with each model. Lastly, we provide a brief discussion on the function-level vs. whole-program vulnerability analysis.

7.1 Error by CWE Distance

According to the Common Weakness Enumeration (CWE) taxonomy, each weakness has an associated level of abstraction: Category, Class, Base, and Variant. These abstraction enables CWEs to be connected on a hierarchical structure, semantically relating two distant CWEs. For instance, CWE-125 (Out-of-Bounds Read) is a Base CWE and falls under the broader CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), a Class-level CWE. In this case, we can consider that CWE-125 and CWE-119 have a hierarchical distance of 1. Following this logic, we can map distance to every CWE pairs, as illustrated in Figure 3, where the hierarchical relationships are quantified.

Based on this hierarchical distance measure between CWEs, we can analyze the relationship between a model's prediction errors and the underlying CWE distance. Naturally, it is expected that the model is likely to misclassify a weakness as another that is semantically similar — i.e., with lower CWE distance. With this intuition, we plot a chart of error counts by CWE distances, for all LLMs we have tested (Figure 4).

Notably, GPT-3.5 and GPT-4 do not exhibit a meaningful correlation between CWE distance and error frequency (note that in Figure 3, there exist more CWE pairs with distances 3 and 4 than distances 1 and 2). In contrast, CodeT5 and CodeBERT demonstrate a clear expected pattern: the model misclassifies more frequently when CWEs are more semantically similar. This suggests that in many cases, these model successfully classify in a semantically

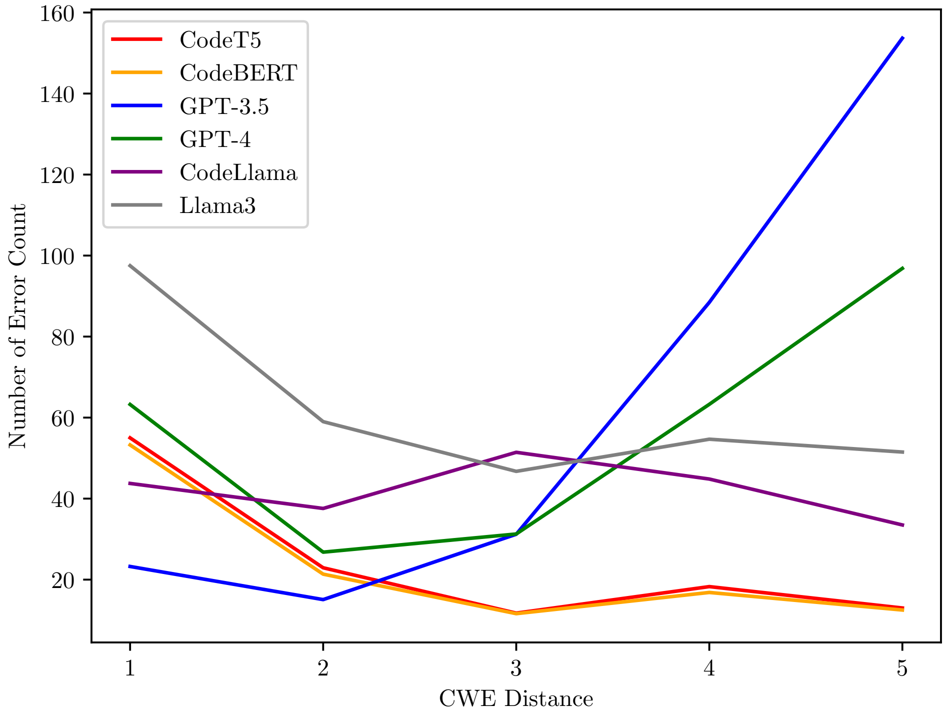

Figure 4: Error Counts by Hierarchical CWE Distance

Table 7: FLOPs amd MACs of models

Model	**CodeT5**	**CodeBERT**	**CodeLlama**	**Llama3**	**GPT**[*]
FLOPs	**2.81E+12**	2.76E+11	3.26E+14	5.79E+10	> 1.26E+24
MACs	**1.41E+12**	1.38E+11	1.64E+14	2.89E+10	> 6.28E+23

[*] Since GPT-3.5 and GPT-4 model's architecture has not been opened, therefore calculation has been estimated with architecture of GPT-3

similar range of weaknesses. We argue that this shows a clear sign that in our experiment, instruct-tuned models show a better performance in Software Vulnerability Identification than online models. Additionally, these findings can imply that incorporating CWE hierarchical information during training may further enhance model performance, by reducing semantic misclassification among closely related weaknesses.

7.2 Computational Cost

Similar to in Section 6.3, we analyze computational cost required to train and test each LLMs. Specifically, we use Floating Point Operations Per Second (FLOPS) [25] to estimate the computational cost of each model. Based on FLOPS, we also estimated the total training time. Additionally, we also estimate Multiply-Accumulate Operations (MACs) as a complementary metric for assessing computational cost required to train and infer on a model. By comparing FLOPS and MACs (as shown in Table 7) alongside the accuracy results of Table 2, we argue that CodeBERT and CodeT5 offers the best trade-off between performance and computational cost for code vulnerability detection.

7.3 Function-level vs. Whole-program Vulnerability Analysis

Our work is based on function-level vulnerability analysis, where the code snippets contains a single function. While most of the previous works follow this practice, there has been a growing need for a whole-program vulnerability analysis. In this subsection, we aim to address the goods and limitations of function-level vulnerability analysis.

While it is easier to collect, process, and train on function-level vulnerable code snippets, we acknowledge that this approach is fundamentally limited by its inability to capture out-of-function behaviors. Function-level analysis lacks visibility into interprocedural context, such as data flow across functions, global state changes, or interactions with external components. To address this, some recent works have explored extending analysis to higher levels of granularity, including interprocedural or whole-program analysis, to capture more complex vulnerability patterns [34, 23].

Nevertheless, we argue that function-level vulnerability analysis can serve as a crucial bridge between research-focused detection methods and real-world, production-level security tools. Especially, under our problem setting where users are limited to use smaller LLMs, connecting detection results from function-level analysis to whole-program analysis would be a crucial next step.

8 Conclusion

The advent of generative AI tools and large language models (LLMs) has led our society to rely on the complex software systems. In this fast-evolving software landscape, ensuring the security and robustness of the software systems has become more critical than ever. While LLM-assisted vulnerability detection methods gives some promises in scalable detection and analysis of underlying codes, they often rely on cloud-based services, raising concerns for developers who prefer to keep their proprietary or sensitive code private during development.

In this paper, we address a situation in which the users aim to automate the process of LLM-based vulnerability identification locally, without uploading or disclosing their source code in development. By utilizing a simple instruction- tuning, we showed that even smaller encoder-decoder LLMs can be proven to be both effective and cost-efficient for identifying software vulnerabilities. Our findings show that with proper tuning and prompt strategies, smaller local models can achieve competitive performance in vulnerability detection, while preserving privacy and intellectual integrity.

Acknowledgments This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (RS-2023-00235509, Development of security monitoring technology based network behavior against encrypted cyber threats in ICT convergence environment)

References

[1] CVE Metrcs – Published CVE Records. https://www.cve.org/About/Metrics. Last Accessed: 04/30/2025.

[2] Common weakness enumeration (cwe). https://cwe.mitre.org/, 2025.

[3] Ömer Aslan, Semih Serkant Aktuğ, Merve Ozkan-Okay, Abdullah Asim Yilmaz, and Erdal Akin. A Comprehensive Review of Cyber Security Vulnerabilities, Threats, Attacks, and Solutions. *Electronics*, 12(6), 2023.

[4] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection, 2023.

[5] Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-Guided LLM Agents for Code Localization, 2025.

[6] Xiaohu Du, Ming Wen, Jiahao Zhu, Zifan Xie, Bin Ji, Huijun Liu, Xuanhua Shi, and Hai Jin. Generalization-Enhanced Code Vulnerability Detection via Multi-Task Instruction Fine-Tuning, 2024.

[7] Matteo Esposito, Valentina Falaschi, and Davide Falessi. An extensive comparison of static application security testing tools. In *EASE*, pages 69–78. ACM, 2024.

[8] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, pages 508–512, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages, 2020.

[10] Michael Fu and Chakkrit Tantithamthavorn. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 608–620, 2022.

[11] Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We? In *APSEC*, pages 632–636. IEEE, 2023.

[12] Katerina Goseva-Popstojanova and Andrei Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Inf. Softw. Technol.*, 68:18–33, 2015.

[13] Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. Repoaudit: An autonomous llm-agent for repository-level code auditing, 2025.

[14] Yuejun Guo, Constantinos Patsakis, Qiang Hu, Qiang Tang, and Fran Casino. Outside the comfort zone: Analysing LLM capabilities in software vulnerability detection. In *ESORICS (1)*, volume 14982 of *Lecture Notes in Computer Science*, pages 271–289. Springer, 2024.

[15] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. Mitigating false positive static analysis warnings: Progress, challenges, and opportunities. *IEEE Trans. Software Eng.*, 49(12):5154–5188, 2023.

[16] Xu Han, Weilin Zhao, Ning Ding, Zhiyuan Liu, and Maosong Sun. PTR: Prompt Tuning with Rules for Text Classification, 2021.

[17] Jingxuan He and Martin T. Vechev. Large language models for code: Security hardening and adversarial testing. In *CCS*, pages 1865–1879. ACM, 2023.

[18] Soolin Kim, Jusop Choi, Muhammad Ejaz Ahmed, Surya Nepal, and Hyoungshick Kim. VulDeBERT: A Vulnerability Detection System Using BERT. In *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 69–74, 2022.

[19] Moohun Lee, Sunghoon Cho, Changbok Jang, Heeyong Park, and Euiin Choi. A rule-based security auditing tool for software vulnerability detection. In *2006 International Conference on Hybrid Information Technology*, volume 2, pages 505–512, 2006.

[20] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimäki, Savanna Lujan, and Fabio Palomba. A critical comparison on six static analysis tools: Detection, agreement, and precision. *J. Syst. Softw.*, 198:111575, 2023.

[21] Jingyue Li, Sindre Beba, and Magnus Melseth Karlsen. Evaluation of open-source IDE plugins for detecting security vulnerabilities. In *EASE*, pages 200–209. ACM, 2019.

[22] Qinbin Li, Junyuan Hong, Chulin Xie, Jeffrey Tan, Rachel Xin, Junyi Hou, Xavier Yin, Zhun Wang, Dan Hendrycks, Zhangyang Wang, Bo Li, Bingsheng He, and Dawn Song. Llm-pbe: Assessing data privacy in large language models, 2024.

[23] Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. On the effectiveness of function-level vulnerability detectors for inter-procedural vulnerabilities. In *ICSE*, pages 157:1–157:12. ACM, 2024.

[24] Syed Shariyar Murtaza, Wael Khreich, Abdelwahab Hamou-Lhadj, and Ayse Basar Bener. Mining trends and patterns of software vulnerabilities. *J. Syst. Softw.*, 117:218–228, 2016.

[25] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM, 2021.

[26] OpenAI. Gpt-3.5 api, 2023. Proprietary model by OpenAI, accessed via API.

[27] OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023. Technical report describing GPT-4, a large-scale multimodal model.

[28] Tosin Daniel Oyetoyan, Bisera Milosheska, Mari Grini, and Daniela Soares Cruzes. Myths and facts about static application security testing tools: An action research at telenor digital. In *XP*, volume 314 of *Lecture Notes in Business Information Processing*, pages 86–103. Springer, 2018.

[29] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 311–318, USA, 2002. Association for Computational Linguistics.

[30] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

[31] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open Foundation Models for Code, 2024.

[32] RunPod. RunPod: Cloud GPU Computing Platform. https://www.runpod.io, 2024.

[33] Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights. *CoRR*, abs/2502.07049, 2025.

[34] Zihua Song, Junfeng Wang, Kaiyuan Yang, and Jigang Wang. Hgivul: Detecting inter-procedural vulnerabilities based on hypergraph convolution. *Inf. Softw. Technol.*, 160:107219, 2023.

[35] Karl Tamberg and Hayretdin Bahsi. Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study. *IEEE Access*, 13:29698–29717, 2025.

[36] Llama Team. The llama 3 herd of models, 2024.

[37] Rahul Telang and Sunil Wattal. An empirical analysis of the impact of software vulnerability announcements on firm stock price. *IEEE Trans. Software Eng.*, 33(8):544–557, 2007.

[38] The MITRE Corporation. CWE Top 25 Most Dangerous Software Weaknesses. Last Accessed: 04/30/2025.

[39] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation, 2021.

[40] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned language models are zero-shot learners. In *ICLR*. OpenReview.net, 2022.

[41] Rajesh Yarra. Llmpatronous: Harnessing the power of llms for vulnerability detection, 2025.

[42] Xin Yin. Pros and cons! evaluating chatgpt on software vulnerability. *CoRR*, abs/2404.03994, 2024.

[43] Xin Yin, Chao Ni, and Shaohua Wang. Multitask-Based Evaluation of Open-Source LLM on Software Vulnerability. *IEEE Trans. Software Eng.*, 50(11):3071–3087, 2024.

[44] Xin Zhou, Ting Zhang, and David Lo. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions, 2024.