VulLibGen: Generating Names of Vulnerability-Affected Packages via a Large Language Model

Anonymous ACL submission

Abstract

Security practitioners maintain reports in vulnerability databases (e.g., GitHub Advisory) to help developers avoid potential risks and deploy vulnerability patches. However, existing work shows that in more than half of the reports, the field of vulnerability-affected packages is missing or incorrect. To help reduce the manual efforts in completing and validating the affected-package field, existing work proposes to automatically identify this information. However, all existing work suffers from low accuracy, relying on relatively small models such as logistic regression and BERT due to linear time cost to the number of packages under consideration. To address these limitations, we propose the first work, a framework named VulLibGen, to explore the use of a large language model (LLM) for directly generating the names of affected packages. VulLibGen conducts supervised fine-tuning (SFT) and retrieval augmented generation (RAG) to supply domain knowledge to the LLM, and a local search technique to ensure that the generated name of an affected package is among the names of the packages under consideration. Our evaluation results show that VulLibGen has an average accuracy of 0.806 for identifying vulnerable packages in the four most popular ecosystems in GitHub Advisory (Java, JS, Python, Go) while the best SOTA ranking approaches achieve only 0.721. Additionally, VulLibGen has provided high value to security practice: we have submitted 28 pairs of <vulnerability, affected package> to GitHub Advisory, and 22 of them have been accepted and merged.

1 Introduction

With the increasing usage of third-party software packages, their security vulnerabilities pose great challenges to software and network systems. A recent study (Wang et al., 2020) shows that 84% third-party packages contain vulnerabilities and

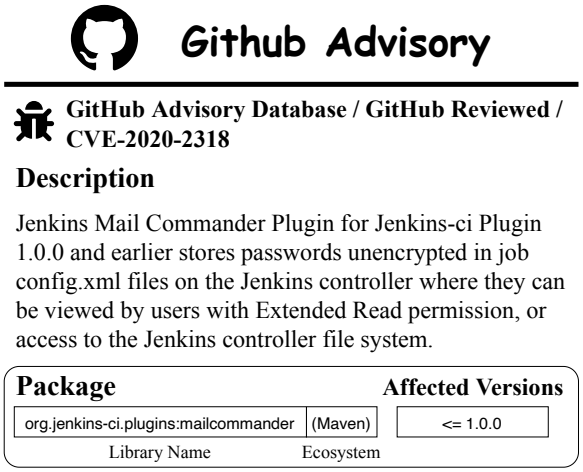

Figure 1: GitHub Advisory Report for CVE-2020-2318

60% of them are high-risk ones. To avoid potential risks posed by these vulnerabilities, security practitioners maintain vulnerability databases for reference, e.g., the National Vulnerability Database (NVD) (COMMERCE, 2024) and GitHub Advisory (GitHub, 2024a). These databases help developers realize and deploy vulnerability patches. Figure 1 shows an example report of one vulnerability, CVE-2020-2318. The vulnerability's affected packages include `org.jenkins-ci.plugins:mailcommander` and its corresponding versions. The field of affected packages is specified by the developers who create this CVE. However, recent studies (Haryono et al., 2022; Dong et al., 2019) show that in more than half of vulnerability reports, this field is missing or incorrect. To alleviate this problem, human maintainers manually complete or validate the affected-package information in databases such as GitHub Advisory (GitHub, 2024b). However, manual completion or validation requires high manual efforts (Haryono et al., 2022; Chen et al., 2023), underscoring the need for automatic identification.

Existing work on automatic identification of vulnerable packages (Chen et al., 2020; Dong et al., 2019; Haryono et al., 2022; Lyu et al., 2023; Chen et al., 2023) suffers from two related limitations

on time cost and accuracy, respectively. First, existing work has linear time cost to the number of packages under consideration (e.g., 435k Java packages) (Lyu et al., 2023; Chen et al., 2023), so that the cost of each model inference has to be kept quite low. Given a vulnerability description (Figure 1), existing work computes a similarity score between the vulnerability description and each package's text description from the ecosystem (e.g., PyPI, Maven), and then ranks all packages based on this score. As a result, the time cost for identifying each vulnerability = the number of candidate packages $\times$ the cost of each model inference (e.g., BERT). Second, existing work suffers from low accuracy due to the adoption of a relatively small model. The first limitation causes the underlying model used for inference to be a relatively small model such as logistic regression and BERT (Chen et al., 2020; Lyu et al., 2023; Haryono et al., 2022; Chen et al., 2023).

To address the preceding limitations, in this paper, we propose the first work, a framework named VulLibGen, to explore the use of a large language model (LLM) for identifying vulnerable packages, given the continuously and rapidly improved effectiveness brought by an LLM for various tasks. VulLibGen uses an LLM to *generate* the names of affected packages instead of ranking the names of packages under consideration. Our rationale for not following the existing work's ranking approaches is that the number (denoted as $|\mathcal{P}|$) of packages under consideration (e.g., 435k Java packages) brings high cost to rank the similarity scores between a vulnerability and each package under consideration. In contrast, in order to identify vulnerable packages, the generative approach invokes the model inference only once, instead of $|\mathcal{P}|$ times.

Specifically, VulLibGen includes two techniques to address two main challenges of generating the names of affected packages. First, we conduct supervised fine-tuning (SFT), in-context learning, and retrieval-augmented generation (RAG) to enhance the domain knowledge of an LLM as an LLM may lack the domain knowledge required to generate the name of the target package. During model inference, the vulnerability description as input fed to an LLM often does not contain the full information of the target package's name, and thus the LLM is required to infer the missing information based on the domain knowledge. Second, we propose a local search technique to ensure that the generated name of an affected package is among the names of the packages under consideration. Recent studies (Vazquez et al., 2023) show that an LLM may generate package names that do not exist. Based on our empirical study of incorrect raw outputs of ChatGPT, we design our local search technique that matches the generation output with the closest package name among the names of the packages under consideration, and produces the matched package name as the final output.

Our evaluation of VulLibGen attains three main findings. First, we observe that the accuracy of VulLibGen significantly outperforms existing approaches (Chen et al., 2020; Haryono et al., 2022; Lyu et al., 2023; Chen et al., 2023) and the computational time cost is comparable. In particular, VulLibGen using Vicuna-13B outperforms the larger ChatGPT and GPT-4 models by employing supervised fine-tuning (SFT). Second, our ablation studies show that SFT, RAG, and local search all help improve the accuracy of VulLibGen. Third, VulLibGen provides high value to security practice: we have submitted 28 pairs of <vulnerability, affected package> to GitHub Advisory, and 22 of them are accepted and merged.

2 Existing Work on Vulnerable Package Identification

Existing Work on Smaller Models. There exist various approaches on vulnerable package identification (Chen et al., 2020; Dong et al., 2019; Haryono et al., 2022; Lyu et al., 2023). However, they all suffer from lower accuracies (Lyu et al., 2023; Chen et al., 2023). Although existing approaches introduce different methods to improve the accuracy, e.g., information extraction (Dong et al., 2019; Anwar et al., 2021) and extreme multi-label classification (Dong et al., 2019; Anwar et al., 2021), all of them rely on smaller language models, e.g., non-neural network models (Chen et al., 2020), logistic regression (Gupta et al., 2021; Lyu et al., 2023) and BERT (Haryono et al., 2022; Chen et al., 2023). The limited model size restricts their capabilities in understanding the text.

Formal Definition of Existing Work. Given a vulnerability description q, the goal of existing work is to select one or more affected packages p from a list $\mathcal{P}$ of existing packages of an ecosystem. At test time, existing work computes the similarity score $sim(p, q)$ for each $p \in \mathcal{P}$, where p is represented by its description documentation in the ecosystem.

Table 1: An Empirical Study on ChatGPT's Incorrect Response in Maven

Error Reason	Example (w/ link)	ChatGPT's Output	Ground Truth (Affected Packages)
Type 1: Incorrect but exist (23% of all errors)	CVE-2015-3158	org.picketlink:picketlink	org.picketlink:picketlink-tomcat-common *Description:* "The invokeNextValve function in *identity/federation/bindings/tomcat/idp/AbstractIDPValve.java* in PicketLink before 2.7.1.Final does not properly check role based authorization, which allows remote authenticated users to gain access to restricted application resources via a (1) direct request . . . "
Type 2: Non-Exist, Partially correct (58% of all errors)	CVE-2011-2730	org.springframework:spring-framework	org.springframework:spring-core *Description:* "VMware SpringSource Spring Framework before 2.5.6.SEC03, 2.5.7.SR023, and 3.x before 3.0.6, when a container supports Expression Language (EL), evaluates EL expressions in tags twice which allows remote attackers to obtain sensitive information. . . . "
	CVE-2020-2167	org.jenkins-ci.plugins:openshift-pipeline	com.openshift.jenkins:openshift-pipeline *Description:* "OpenShift Pipeline Plugin 1.0.56 and earlier does not configure its YAML parser to prevent the instantiation of arbitrary types. This results in a remote code execution (RCE) vulnerability exploitable by users able to provide YAML input files to OpenShift Pipeline Plugin's build step. . . . "
Type 3: Non-Exist, Completely incorrect (19% of all errors)	CVE-2020-11974	mysql:mysql-connector-java	org.apache.dolphinscheduler:dolphinscheduler *Description:* "In DolphinScheduler 1.2.0 and 1.2.1, with mysql connector a remote code execution vulnerab -ility exists when choosing mysql as database."
	CVE-2019-13234	N/A	org.opencms:opencms-core *Description:* "In the Alkacon OpenCms Apollo Template 10.5.4 and 10.5.5 there is XSS in the search engine."

As a result, the time cost on each vulnerability = $|\mathcal{P}|\times$ the cost to compute each similarity score.

Existing Work's Efforts on Scaling to Larger Models. To improve the accuracy, existing work leverages re-ranking with the BERT model (Chen et al., 2023); nevertheless, there remains a large room for improvement. More specifically, they first use TF-IDF to rank all packages in the ecosystem (435k in Java and 506k in Python), then re-rank all top-512 packages using BERT. While the recall@512 of TF-IDF is 0.9, their Accuracy@1 is far from 0.9.

3 Two Challenges with LLM Generation

The Generative Approach. To investigate the potential of using even larger models to improve the accuracy, we propose a framework that uses LLMs (7B or larger) to *generate* instead of ranking the affected packages. We cannot adopt the rank approach in existing work (Section 2) since the cost to compute each similarity and $|\mathcal{P}|$ are very high.

Formally speaking, given the vulnerability description q, the generative approach directly generates the affected package names p, therefore the time cost on each vulnerability = $1\times$the inference cost of the LLM. Nevertheless, there exist two challenges in the generative approach.

Challenge 1: Lack of Domain Knowledge. The first challenge is that there may exist a knowledge gap for the LLM to generate the correct package. This is because the description may not contain the full information about the affected package name. For example, CVE-2020-2167 in Table 1 is about the Java package `com.openshift.jenkins.openshift-pipeline`, but the the description does not mention the word "*Jenkins*". To predict the correct package name, the LLM has to rely on domain knowledge to complete this information. Existing work have used various methods to bridge the knowledge gap of LLMs, e.g., supervised fine-tuning (Prottasha et al., 2022; Church et al., 2021) and retrieval augmented generation (Lewis et al., 2020; Mao et al., 2020; Liu et al., 2020; Cai et al., 2022).

Challenge 2: Generating Non-Existing Package Names. Following a previous studies on Reddit[1], the second challenge is that LLM may generate library names that do not exist. Existing work has adopted post-processing to alleviate this problem (Jin et al., 2023; Roziere et al., 2021). Following existing work, we can potentially leverage post-processing by matching the generated package with the closest existing package based on their edit distance.

To understand whether post-processing is promising for solving Challenge 2 and to study how to design the post-processing algorithm, we conduct an empirical study on ChatGPT's incorrect response, the study result can be seen in Table 1. The study uses 2,789 Java vulnerability descriptions collected in a recent work (Chen et al., 2023). We divide all ChatGPT responses into four types: 1. the package is correct (42%); 2. the package is incorrect but it exists (13%); 3. the package does not exist and is partially correct (34%); 4. the package is completely incorrect (11%).

From the study result, we draw the conclusion that post-processing using edit-distance matching is a promising approach to solve Challenge 2. Among

[1] https://www.reddit.com/r/ChatGPT/comments/zneqyp/chatgpt_hallucinates_a_software_library_that/

the three types of errors, post-processing can help with Type 2 errors, which constitute 58% errors. The average edit distance between ChatGPT output and the ground truth in Type 2 is 76% on average which shows promise in improving the accuracy with post-processing. By applying a naive edit-distance matching on the ChatGPT output, the accuracy is improved from 42% to 51%.

4 VulLibGen Framework

To address the two challenges in LLM generation, we employ the following techniques: first, we use supervised fine-tuning or in-context learning to enhance the domain knowledge in LLM; second, we further employ the retrieval-augmented framework (RAG) to enhance the knowledge when SFT is not easy; third, we design a local search technique which alleviates the non-existing package name problem. The VulLibGen framework can be found in Figure 2. [2].

4.1 Supervised Fine-Tuning/In-Context Learning

To solve the first challenge (Section 3), we incorporate supervised fine-tuning (Prottasha et al., 2022; Church et al., 2021) and in-context learning (Dong et al., 2022; Olsson et al., 2022) in VulLibGen. For SFT, we use the full training data (Table 2); for ICL, we randomly sample 3 examples from the training data for each evaluation vulnerability. For both SFT and ICL, the input and output of the LLM follow the following format: Input: the same prompt as Figure 2 [2], Output: "*The affected package is [package name]*". The hyper-parameters used for ICL and SFT are listed in Table 7 of Appendix 10.2.

4.2 Retrieval-Augmented Generation (RAG)

To further enhance the LLM's domain knowledge especially when SFT is not easy (e.g., ChatGPT and GPT4), we employ retrieval-augmented generation (RAG) in VulLibGen.

Retriever setting. Given the description of a vulnerability, our retriever ranks unique package names (Table 2) based on the similarity score between the vulnerability description and the package description. The descriptions of Java, JavaScript, Python, and Go packages are obtained from Maven [3], NPM [4], Pypi [5], and Go [6] documentations. For example, the description of the package `org.jenkins-ci.plugins:mailcommander` is *"This plug-in provides function that read a mail subject as a CLI Command."*. Our retriever follows (Chen et al., 2023)'s re-ranking strategy, i.e., first rank all packages (e.g., 435k in Java) using TF-IDF, then re-rank the top 512 packages using a BERT-base model fine-tuned on the same training data in Table 2.

4.3 Local Search

To solve the second challenge (Section 3), we incorporate post-processing in VulLibGen. Based on the empirical study results in Section 3, we design a local search technique to match the generation output with the closest package name from an existing package list (Algorithm 1 in Appendix 10.1).

Algorithm 1 employs the edit distance as the metric and respects the structure of the package name. Formally, a package name can be divided into two parts: its prefix and suffix (separated by a special character, e.g., ':' in Java). The prefix (e.g., the artifact ID of Java packages) specifies the maintainer/group of this package, and the suffix (e.g., the group ID of Java packages) specifies the functionalities of this package. Specifically, Java, Go, and part of JS packages can be explicitly divided while Python and the rest of JS packages only specify their functionalities in their names. We denote the prefix of a package name as empty if it can not be divided.

Algorithm 1 first compares the generated suffix with all existing suffix names and matches the suffix to the closest one. After fixing the suffix, we can then obtain the list of prefixes that co-occur at least once with this suffix. We match the generated prefix with the closest prefix in this list. The reason that we opt to match the suffix first is twofold. First, our pilot study shows that the vulnerability description more frequently mentions the suffix than the prefix. Among all 2,789 vulnerabilities investigated in Section 3, their description mentions 12.4% of the tokens in the prefixes of the affected packages and 66.0% of the tokens in their suffixes of the affected packages; second, our study also shows that each suffix co-occurs with fewer unique prefixes

[2] Our prompt in Figure 2 is: "*Below is a [Programming Language] vulnerability description. Please identify the software name affected by it. Input: [DESCRIPTION]. The top k search results are: $[L_1][L_2]\cdots[L_k]$. Please output the package name in the format "ecosystem:library name". ### Response: The affected packages:*".

[3] https://mvnrepository.com

[4] https://www.npmjs.com

[5] https://pypi.org

[6] https://pkg.go.dev

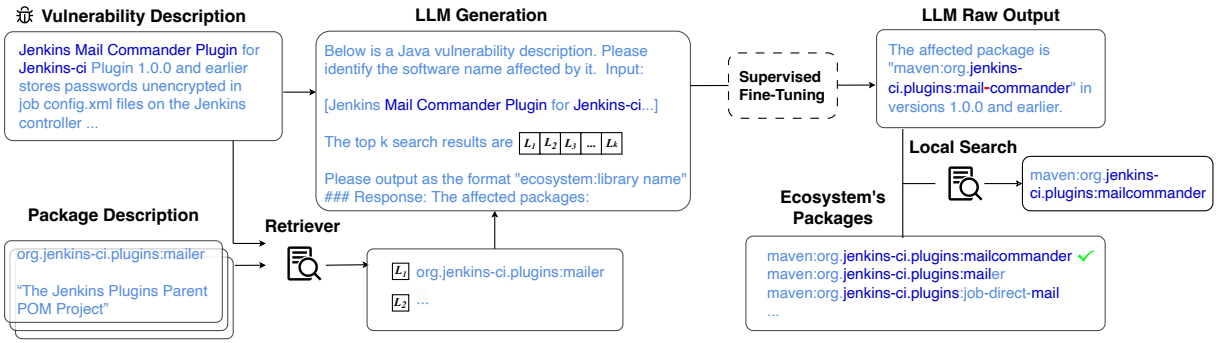

Figure 2: The VulLibGen Framework

than conversely. In all 435k Java packages, each prefix has 5.86 co-occurred suffixes while each suffix has only 1.13 co-occurred prefixes on average. As a result, for some vulnerabilities, it is easier to identify the prefix by first matching the suffix, and then matching the suffix with the co-occurred prefix list.

5 Evaluation

5.1 Evaluation Setup

Dataset. In this paper, we evaluate the effectiveness of VulLibGen using the GitHub Advisory database, since each vulnerability in GitHub Advisory is manually reviewed and verified by expert maintainers (GitHub, 2024b). In GitHub Advisory, the vulnerabilities are classified by the associated Programming languages (GitHub, 2024a), therefore we can construct a list of programming language-focused datasets.

Our dataset focuses on four widely used programming languages: Java[7], JavaScript (JS), Python, and Go. The statistics of our dataset are listed in Table 2. In total, our dataset includes 2,789 Java, 3,193 JS, 2,237 Python, and 1,351 Go vulnerabilities, respectively. To the best of our knowledge, this is the first dataset for identifying vulnerable packages with various programming languages. For each PL, we split the train/validation/test data with the 3:1:1 ratio. The split is in chronological order to simulate a more realistic scenario and to prevent lookahead bias (Kenton, 2024).

Comparative Methods. To evaluate the effectiveness of VulLibGen, we contrast it with four SOTA ranking approaches, FastXML (Chen et al., 2020), LightXML (Haryono et al., 2022), Chronos (Lyu et al., 2023), and VulLibMiner (Chen et al., 2023) for comparison. Recent studies (Lyu et al., 2023; Chen et al., 2023) show that they outperform other approaches, such as Bonsai (Khandagale et al., 2020) and ExtremeText (Wydmuch et al., 2018).

[7]For Java, we use VulLib (Chen et al., 2023), which is expanded from the Java vulnerabilities in GitHub Advisory.

Table 2: The Statistics of the GitHub Advisory Dataset

	Java	JS	Python	Go
#Vulnerabilities:				
Training	1,668	1,915	1,342	810
Validation	556	639	447	270
Testing	565	639	448	271
Total	2,789	3,193	2,237	1,351
#Unique packages in the dataset:				
	2,095	2,335	710	601
#Total packages in their ecosystems:				
	435k	2,551k	507k	12k
#Avg. tokens of packages:				
	13.44	4.56	3.96	8.24

Models in VulLibGen. The models we evaluate for the VulLibGen framework include both commercial LLMs, e.g., ChatGPT (gpt-3.5-turbo) and GPT4 (gpt-4-1106-preview), and open-source LLMs, e.g., LLaMa (Touvron et al., 2023) and Vicuna (Chiang et al., 2023).

We assess open-source LLMs in two scenarios: few-shot in-context learning using 3 examples randomly sampled from the training data and supervised fine-tuning using the full training data. For the open-source LLMs (Table 3), we use ICL/SFT + RAG + local search, whereas for commercial LLMs, we use RAG + local search only.

Evaluation Environments Our evaluations are conducted on the system of Ubuntu 18.04. We use one Intel(R) Xeon(R) Gold 6248R@3.00GHz CPU, which contains 64 cores and 512GB memory. We use 8 Tesla A100 PCIe GPUs with 40GB memory for model training and inference. In total, our experiments constitute 200 GPU days (32 groups in RQ1 + 68 groups in RQ2, and each group costs 0.25 GPU days across 8 GPUs).

Metrics Since more than 60% of the vulnerabilities affect only one package (Chen et al., 2023), we

Table 3: VulLibGen's Accuracy@1 with Various LLMs

Approach	Java	JS	Python	Go	Avg.
Ranking-based Non-LLMs:					
FastXML	0.292	0.078	0.491	0.277	0.285
LightXML	0.450	0.146	0.529	0.494	0.405
Chronos	0.516	0.447	0.550	0.710	0.556
VulLibMiner	0.669	0.742	0.825	0.647	0.721
Commercial LLMs:					
ChatGPT	0.758	0.732	0.915	0.646	0.763
GPT4	**0.783**	0.768	0.868	0.712	0.783
Few-Shot ICL on Open-Source LLMs:					
LLaMa-7B	0.002	0.237	0.036	0.000	0.069
LLaMa-13B	0.122	0.238	0.049	0.048	0.114
Vicuna-7B	0.110	0.495	0.694	0.428	0.432
Vicuna-13B	0.186	0.513	0.527	0.394	0.405
Full SFT on Open-Source LLMs:					
LLaMa-7B	0.710	**0.773**	0.924	0.716	0.781
LLaMa-13B	0.720	0.765	0.904	0.775	0.791
Vicuna-7B	0.697	0.768	0.929	0.782	0.794
Vicuna-13B	0.710	**0.773**	**0.935**	**0.804**	**0.806**

use Accuracy@1 for evaluating a model. That is, exact match between the first generation or ranking output and the ground truth.

5.2 Evaluation of VulLibGen

In this section, we evaluate the effectiveness of VulLibGen. We seek to answer the following research question: How does VulLibGen compare to existing work on identifying vulnerable packages?

Overall Accuracy: Existing Work vs. VulLibGen. From Table 3 we can observe that for all programming languages, VulLibGen achieves substantially higher accuracies compared to existing work. As a result, by leveraging LLMs, VulLibGen can effectively generate the names of affected packages with high accuracies.

Overall, VulLibGen using supervised fine-tuning on the Vicuna-13B model has the best performance. Fine-tuning Vicuna-13B even outperforms the larger ChatGPT and GPT4 models on all datasets besides Java. As a result, the knowledge gap of LLMs can be effectively bridged by leveraging supervised fine-tuning. We further conduct statistical significance tests (Kim, 2015) between the best-performing generative approach (i.e., VulLibGen using Vicuna-13B SFT) and the best-performing existing work (i.e., VulLibMiner (Chen et al., 2023)). The p-values of all tests are smaller than $1e$-5, and the detailed results is listed in Appendix (Table 8).

When Is VulLibGen More Advantageous? From Table 3 observe that the gap between the best-performing VulLibGen method and the best-performing existing approach for each programming language are 0.041, 0.031, 0.11, and 0.157. By comparing with the data statistics in Table 2, we can see that this gap is highly correlated with *#Unique packages in the dataset* and *#Total packages in the ecosystem*. In general, VulLibGen is less advantageous when the output package name is longer and has a larger token space.

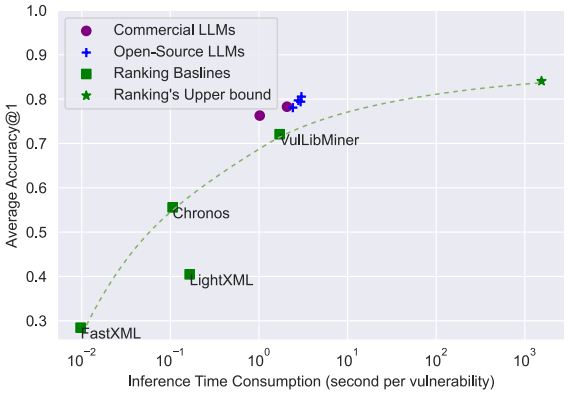

Figure 3: Trade-Offs between Efficiency and Accuracy

Efficiency of Existing Work vs. VulLibGen. In Figure 3, we visualize the actual computational cost and accuracy of each method in Table 3. We further mark the upper bound of the ranking-based approach using LLMs for comparison. The Accuracy@1 is upper-bounded by the recall@512 of TF-IDF, i.e., the best possible Accuracy@1; while the time cost is upper-bounded by the time cost of invoking the 13B model 512 times (10 mins).

We can observe the VulLibGen achieves a sweet spot in the effectiveness and efficiency trade-off. When compared with existing work, VulLibGen achieves better a Accuracy@1 while the time cost is comparable to the best-performed existing work (Chen et al., 2023). When compared with the upper bound, VulLibGen achieves a slightly lower Accuracy@1 while consuming less than 1/100 time and computation resources.

5.3 Ablation Studies on VulLibGen

In this sub section, we conduct ablation studies on the three components of VulLibGen: supervised fine-tuning, RAG, and local search.

SFT's Improvement. By comparing the results of in-context learning vs supervised fine-tuning in Table 3, we can see that SFT outperforms ICL by a larger margin. We observe that SFT also outperforms the baseline with neither ICL nor SFT, we have not shown the latter result due to space limit. This result indicates that for the 7B and 13B models, supervised fine-tuning on the full training data

Table 4: RAG's Improvement ($Accuracy@1_{RAG} - Accuracy@1_{Raw}$)

Language	Java	JS	Python	Go
Commercial LLMs:				
ChatGPT	16.1% ↑	3.6% ↑	38.8% ↑	57.6% ↑
GPT4	12.1% ↑	0.9% ↑	0.9% ↑	31.0% ↑
Full SFT on Open-Source LLMs:				
LLaMa-7B	2.2% ↑	2.2% ↑	3.1% ↑	1.8% ↓
LLaMa-13B	3.3% ↑	0.4% ↑	2.0% ↑	3.7% ↑
Vicuna-7B	13.6% ↑	2.3% ↑	3.8% ↑	3.7% ↑
Vicuna-13B	8.7% ↑	1.2% ↑	4.9% ↑	0.0% -
Average	9.3% ↑	1.8% ↑	8.9% ↑	15.7% ↑

is essential in bridging the models' knowledge gap.

RAG's Overall Improvement: Table 4 shows the improvement of our RAG technique in Accuracy@1. Specifically, it improves the Accuracy@1 by 9.3%, 1.8%, 8.9%, and 15.7% on each programming language, respectively. These improvements indicate that our RAG technique is effective in helping generate the names of vulnerable packages.

Table 4 also indicates that RAG's improvement in commercial LLMs is higher than that of open-source LLMs. Especially in Go vulnerabilities, our RAG technique improves the Accuracy@1 by 57.6% and 31.0% on ChatGPT and GPT4. The main reason is that both ChatGPT and GPT4 do not have sufficient domain knowledge about Go packages as they are relatively newer than packages of other programming languages (Hall, 2023).

RAG Improvement vs. k/Retrieval Algorithm. We evaluate whether k and the retrieval algorithm affect the end-to-end effectiveness of VulLibGen. Specifically, we focus on Java vulnerabilities (as Java package names are the most difficult to generate). The result can be found in Table 5.

For k, we conduct an Analysis of Variance (ANOVA) (St et al., 1989) among the Accuracy@1 of six representative numbers of RAG packages (ranging from 1 to 20). Although $k = 20$ has a slightly higher accuracy than $k = 1$ for both TF-IDF and BERT, this difference is not significant. In fact, the t-test results show that there is no significant difference among the Accuracy@1 of different k values ($p = 0.814$ for TF-IDF and $p = 0.985$ for BERT).

As for the retrieval algorithm, we observe that Accuracy@1 with TF-IDF results is quite similar to that of non-RAG inputs, and the Accuracy@1 with BERT results is substantially higher than that of non-RAG/TF-IDF results. As a result, it is essential to use BERT-retrieved results in RAG.

Local Search's Improvement. Table 6 shows the end-to-end improvement in Accuracy@1 of VulLibGen before and after local search. Our local search technique improves the Accuracy@1 by 3.43%, 1.02%, 1.57%, and 6.20% on each programming language. Additionally, it is more effective on commercial LLMs (an average improvement of 4.58%) than fine-tuned open-source LLMs (an average improvement of 2.29%). Since commercial LLMs are not fine-tuned, local search plays an important role in improving the effectiveness of generation.

5.4 Evaluating VulLibGen Performance in Real World Setting

To examine VulLibGen's performance in the real-world setting, we randomly sample a subset of the vulnerability descriptions in Java and JS. We use VulLibGen to generate the package names and submit the generated names (VulLibGen with Vicuna-13B) to GitHub Advisory.

We report 28 pairs of <vulnerability, affected package> that are not listed in GitHub Advisory. At the time of the writing, 22 of them have been accepted and merged into GitHub Advisory. Among the rest 6 packages, 3 of them are considered non-vulnerabilities, and 3 of them are considered incorrect affected packages. The details of these issues are listed in the Appendix (Table 9).

This result highlights the real-world performance of VulLibGen in automatically identifying affected package names.

6 Related Work

Vulnerable Package/Version Identification. There exist numerous works on identifying the affected software package and versions. Multiple existing works model this problem as an NER problem, i.e., extracting the subset of description about the package (Dong et al., 2019; Anwar et al., 2021; Jo et al., 2022; Kuehn et al., 2021; Yang et al., 2021) or version (Dong et al., 2019; Zhan et al., 2021; Zhang et al., 2019; Backes et al., 2016; Zhang et al., 2018; Tang et al., 2022; Gorla et al., 2014; Wu et al., 2023). The NER approach works well for the version identification since many version numbers are already in the description (Dong et al., 2019). As the package names are often only partially mentioned (e.g., CVE-2020-2167 in Table 1), the NER approaches are less effective (Lyu et al., 2023). Another

Table 5: Accuracy@1 with Various RAG Inputs in Generating the Names of Java Affected Packages

IR Model:	None	TF-IDF Results						BERT Results					
#RAG packages:		1	2	3	5	10	20	1	2	3	5	10	20
Commercial LLMs:													
ChatGPT	0.597	0.523	0.498	0.508	0.552	0.540	0.567	0.758	0.743	0.722	0.718	0.715	0.710
GPT4	0.676	0.619	0.559	0.588	0.619	0.626	0.638	0.783	0.773	0.784	0.792	0.797	0.792
Full SFT on Open-Source LLMs:													
LLaMa-7B	0.688	0.692	0.697	0.701	0.563	0.591	0.609	0.710	0.701	0.710	0.678	0.665	0.683
LLaMa-13B	0.687	0.688	0.687	0.696	0.653	0.635	0.623	0.720	0.702	0.701	0.701	0.704	0.703
Vicuna-7B	0.561	0.596	0.398	0.404	0.441	0.439	0.421	0.697	0.701	0.683	0.685	0.706	0.683
Vicuna-13B	0.623	0.609	0.450	0.418	0.650	0.655	0.680	0.710	0.712	0.701	0.722	0.719	0.720

Table 6: Local Search's Improvement ($Accuracy@1_{Search} - Accuracy@1_{Raw}$)

Language	Java	JS	Python	Go
Commercial LLMs:				
ChatGPT	4.1% ↑	1.2% ↑	0.7% ↑	11.1% ↑
GPT4	5.3% ↑	2.5% ↑	2.9% ↑	8.8% ↑
Full SFT on Open-Source LLMs:				
LLaMa-7B	2.9% ↑	0.9% ↑	2.2% ↑	7.0% ↑
LLaMa-13B	3.3% ↑	0.3% ↑	0.7% ↑	4.1% ↑
Vicuna-7B	3.9% ↑	0.9% ↑	1.8% ↑	3.3% ↑
Vicuna-13B	1.1% ↑	0.3% ↑	1.1% ↑	2.9% ↑
Average	3.4% ↑	1.0% ↑	1.4% ↑	6.2% ↑

branch of work models the package identification problem as extreme multi-label learning (XML) where each package is a class (Chen et al., 2020; Haryono et al., 2022; Lyu et al., 2023). However, these methods are limited to less than 3k classes (the labels in their dataset). Finally, (Chen et al., 2023) leverages the re-ranking approach using BERT; however, there still exists a gap between their method's accuracy and the best possible performance (Table 3).

Retrieval vs Generation. Existing work has investigated scenarios of replacing the retrieval with generation. For example, Yu et al. (Yu et al., 2022) leverages LLM to generate the context documents for question answering, rather than retrieving them from a text corpus. Their experiment shows that the generative approach has a comparable performance to the retrieval approach on the QA task. However, since our task requires us to generate the exact package name, their conclusion is not directly transferrable to our task.

Retrieval-Augmented Generation Retrieval-augmented generation (RAG) (Lewis et al., 2020; Mao et al., 2020; Liu et al., 2020; Cai et al., 2022) is a widely used technique and has shown its effectiveness in various generation tasks, e.g., code generation or question answering. Specifically, RAG enhances the performance of a generative model by incorporating knowledge from a database so that LLMs can extract and comprehend correct domain knowledge from the RAG inputs.

Reducing Hallucination. In Section 3, we show that ChatGPT's raw output package name may not exist. This phenomenon is similar to hallucination (Ji et al., 2023; Tonmoy et al., 2024), which occurs in various LLM-related tasks. Among hallucination reduction approaches, post-processing (Madaan et al., 2023; Kang et al., 2023) is a widely used one. For example, in code generation tasks, existing work (Jin et al., 2023; Chen et al., 2022; Zhang et al., 2023; Huynh Nguyen et al., 2022) adopts post-processing techniques to reduce/rerank programs generated by LLMs, e.g., using deep-learning models, test cases, or compilers to determine whether a generated program is correct and remove incorrect programs. However, such techniques cannot be directly adopted in our task because validating the generated names of affected packages is relatively difficult. It requires a Proof-of-Chain (PoC) (Mosakheil, 2018), which is often unavailable due to security concerns. Therefore, we design our local search algorithm focusing on Type 2 errors in Table 1.

7 Conclusion

In this paper, we have proposed VulLibGen, the first framework for identifying vulnerable packages using LLM generation. VulLibGen conducts retrieval-augmented generation, supervised fine-tuning, and a local search technique to improve the generation. VulLibGen is highly effective, achieving an accuracy of 0.806 while the best SOTA approaches achieve only 0.721. VulLibGen has shown high value to security practice. We have submitted 28 pairs of <vulnerability, affected package> to GitHub advisory, and 22 of them have been accepted and merged.

8 Limitation

Our work has several limitations, which we plan to address in our future work:

Improving the Generation of Complicated Languages. As discussed in Section 5.2, the effectiveness of VulLibGen highly depends on the token length and the number of unique packages. For example, Vicuna-13B's Accuracy@1 in Java vulnerabilities (0.710) is less than that of Python vulnerabilities (0.935). To improve the generation accuracy of complicated languages such as Java, we plan to further enhance the knowledge of LLM using techniques such as constrained decoding (Post and Vilar, 2018). We leave this as our future work.

Generating Package Names with Limited Ecosystem Knowledge. Though VulLibGen has demonstrated its effectiveness in four widely-used programming languages, some other programming languages, e.g., C/C++, do not have a commonly used ecosystem that maintains all its packages. Thus, it is difficult to generate/retrieve the affected packages of C/C++ vulnerabilities as we do not have specific ranges during the RAG step of VulLibGen. Exploring how to generate RAG results without a commonly used ecosystem (e.g., Maven or Pypi) or collecting other useful information for RAG is the future work of this paper.

9 Ethical Consideration

License/Copyright. VulLibGen utilizes open-source data from GitHub Advisory, along with four third-party package ecosystems. We refer users to the original licenses accompanying the resources of these data.

Intended Use. VulLibGen is designed as an automatic tool to assist maintainers of vulnerability databases, e.g., GitHub Advisory. Specifically, VulLibGen helps generate the names of affected packages to complement the missing data of these databases. The usage of VulLibGen is also illustrated in Section 4 and our intended usage of VulLibGen is consistent with that of GitHub Advisory (GitHub, 2024a).

Potential Misuse. Similar to existing open-source LLMs, one potential misuse of VulLibGen is generating harmful content. Considering that we use open-source vulnerability data for LLM fine-tuning, the LLM might view harmful content during this step. To avoid harmful content, we use only reviewed vulnerability data in GitHub Advisory, so such misuse will unlikely happen. Overall, the scientific and social benefits of the research arguably outweigh the small risk of their misuse.

References

Afsah Anwar, Ahmed Abusnaina, Songqing Chen, Frank Li, and David Mohaisen. 2021. Cleaning the nvd: Comprehensive quality assessment, improvements, and analyses. *IEEE Transactions on Dependable and Secure Computing*, 19(6):4255–4269.

Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367.

Deng Cai, Yan Wang, Lemao Liu, and Shuming Shi. 2022. Recent advances in retrieval-augmented text generation. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 3417–3419.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Tianyu Chen, Lin Li, Bingjie Shan, Guangtai Liang, Ding Li, Qianxiang Wang, and Tao Xie. 2023. Identifying vulnerable third-party libraries from textual descriptions of vulnerabilities and libraries.

Yang Chen, Andrew E Santosa, Asankhaya Sharma, and David Lo. 2020. Automated identification of libraries from vulnerability data. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, pages 90–99.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E Gonzalez, et al. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality. *See https://vicuna. lmsys. org (accessed 14 April 2023)*.

Kenneth Ward Church, Zeyu Chen, and Yanjun Ma. 2021. Emerging trends: A gentle introduction to fine-tuning. *Natural Language Engineering*, 27(6):763–778.

U.S. DEPARTMENT OF COMMERCE. 2024. Nvd.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey for in-context learning. *arXiv preprint arXiv:2301.00234*.

Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. 2019. Towards the detection of inconsistencies in public security vulnerability reports. In *28th USENIX security symposium (USENIX Security 19)*, pages 869–885.

GitHub. 2024a. Github-advisory.

GitHub. 2024b. Github-advisory-review.

Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *Proceedings of the 36th international conference on software engineering*, pages 1025–1035.

Nilesh Gupta, Sakina Bohra, Yashoteja Prabhu, Saurabh Purohit, and Manik Varma. 2021. Generalized zero-shot extreme multi-label learning. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 527–535.

Jonathan Hall. 2023. How well does chatgpt understand go?

Stefanus A Haryono, Hong Jin Kang, Abhishek Sharma, Asankhaya Sharma, Andrew Santosa, Ang Ming Yi, and David Lo. 2022. Automated identification of libraries from vulnerability data: Can we do better?

Minh Huynh Nguyen, Nghi DQ Bui, Truong Son Hy, Long Tran-Thanh, and Tien N Nguyen. 2022. Hierarchynet: Learning to summarize source code with heterogeneous representations. *arXiv e-prints*, pages arXiv–2205.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38.

Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. *arXiv preprint arXiv:2303.07263*.

Hyeonseong Jo, Yongjae Lee, and Seungwon Shin. 2022. Vulcan: Automatic extraction and analysis of cyber threat intelligence from unstructured text. *Computers & Security*, 120:102763.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2312–2323. IEEE.

Will Kenton. 2024. Look-ahead bias: What it means, how it works.

Sujay Khandagale, Han Xiao, and Rohit Babbar. 2020. Bonsai: diverse and shallow trees for extreme multi-label classification. *Machine Learning*, 109(11):2099–2119.

Tae Kyun Kim. 2015. T test as a parametric statistic. *Korean journal of anesthesiology*, 68(6):540–546.

Philipp Kuehn, Markus Bayer, Marc Wendelborn, and Christian Reuter. 2021. Ovana: An approach to analyze and improve the information quality of vulnerability databases. In *Proceedings of the 16th International Conference on Availability, Reliability and Security*, pages 1–11.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jingkai Siow, and Yang Liu. 2020. Retrieval-augmented generation for code summarization via hybrid gnn. *arXiv preprint arXiv:2006.05405*.

Yunbo Lyu, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Zhipeng Zhao, Xuan-Bach D Le, Ming Li, and David Lo. 2023. Chronos: Time-aware zero-shot identification of libraries from vulnerability reports. *arXiv preprint arXiv:2301.03944*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*.

Yuning Mao, Pengcheng He, Xiaodong Liu, Yelong Shen, Jianfeng Gao, Jiawei Han, and Weizhu Chen. 2020. Generation-augmented retrieval for open-domain question answering. *arXiv preprint arXiv:2009.08553*.

Jamal Hayat Mosakheil. 2018. Security threats classification in blockchains.

Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, et al. 2022. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*.

Matt Post and David Vilar. 2018. Fast lexically constrained decoding with dynamic beam allocation for neural machine translation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*.

Nusrat Jahan Prottasha, Abdullah As Sami, Md Kowsher, Saydul Akbar Murad, Anupam Kumar Bairagi, Mehedi Masud, and Mohammed Baz. 2022. Transfer learning for sentiment analysis using bert based supervised fine-tuning. *Sensors*, 22(11):4157.

Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*.

Lars St, Svante Wold, et al. 1989. Analysis of variance (anova). *Chemometrics and intelligent laboratory systems*, 6(4):259–272.

Wei Tang, Yanlin Wang, Hongyu Zhang, Shi Han, Ping Luo, and Dongmei Zhang. 2022. Libdb: An effective and efficient framework for detecting third-party libraries in binaries. *arXiv preprint arXiv:2204.10232*.

SM Tonmoy, SM Zaman, Vinija Jain, Anku Rani, Vipula Rawte, Aman Chadha, and Amitava Das. 2024. A comprehensive survey of hallucination mitigation techniques in large language models. *arXiv preprint arXiv:2401.01313*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Hernan C Vazquez, J Andres Diaz-Pace, and Antonela Tommasel. 2023. The javascript package selection task: A comparative experiment using chatgpt. In *2023 XLIX Latin American Computer Conference (CLEI)*, pages 1–10. IEEE.

Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45. IEEE.

Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. 2023. Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1046–1058. IEEE.

Marek Wydmuch, Kalina Jasinska, Mikhail Kuznetsov, Róbert Busa-Fekete, and Krzysztof Dembczynski. 2018. A no-regret generalization of hierarchical softmax to extreme multi-label classification. *Advances in neural information processing systems*, 31.

Guanqun Yang, Shay Dineen, Zhipeng Lin, and Xueqing Liu. 2021. Few-sample named entity recognition for security vulnerability reports by fine-tuning pre-trained language models. In *Deployable Machine Learning for Security Defense: Second International Workshop, MLHat 2021, Virtual Event, August 15, 2021, Proceedings 2*.

Wenhao Yu, Dan Iter, Shuohang Wang, Yichong Xu, Mingxuan Ju, Soumya Sanyal, Chenguang Zhu, Michael Zeng, and Meng Jiang. 2022. Generate rather than retrieve: Large language models are strong context generators. *arXiv preprint arXiv:2209.10063*.

Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1695–1707. IEEE.

Jiexin Zhang, Alastair R Beresford, and Stephan A Kollmann. 2019. Libid: reliable identification of obfuscated third-party android libraries. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 55–65.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. *arXiv preprint arXiv:2305.04087*.

Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. 2018. Detecting third-party libraries in android applications with high precision and recall. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 141–152. IEEE.

10 Appendix

10.1 Our Local Search Algorithm

Algorithm 1: Local Search

Input : $rawName$, a generated package name
Output : $vulnNames$, names of affected packages.

```
// Pre-process on name list
1  nameDict, suffixes =← {}, ∅;
2  for name ∈ nameList do
3      prefix, suffix ← name.split("/:");
4      nameDict[suffix].add(prefix);
// Search the closest prefix/suffix
5  prefix, suffix ← rawName.split("/:");
6  edit.weight ← (W_insert, W_delete, W_replace);
7  suffix' ← argmin_{s∈suffixes} edit(suffix, s);
8  prefixes ← nameDict[suffix'];
9  if prefixes.isEmpty() then
10     return {suffix'};
11 else
12     prefix' ← argmin_{p∈prefixes} edit(prefix, p);
13     return {prefix'} : {suffix'};
```

The pseudocode of our local search algorithm is shown in Algorithm 1. The input of this algorithm includes one package name generated by LLM together with the name list of existing libraries under the same ecosystem. The output of this algorithm is the name of one existing package that is the closest to the generated package name.

In Lines 1-4, we pre-process the name list of candidate packages. Now that we divide a package

name into its prefix and suffix, we first construct the dictionary $nameDict$ that maps a suffix into its corresponding prefix.

In Lines 5-13, we search for the closest package name of the input package name, $rawName$. In Line 7, we use its suffix, $suffix$ to find its closest and existing suffix, $suffix'$. Then in Lines 8-13, we first determine whether it contains a corresponding prefix. If it has no prefix (e.g., a Python package), we directly return the closest suffix. Otherwise, we find its closest prefix, $prefix'$, from all prefixs that correspond to $suffix'$. Additionally, in Line 6, we manually set the weight used in calculating edit distances because LLMs change the package names in terms of tokens instead of characters. Thus, the weight of inserting one character should be smaller than that of deleting and replacing one, and we set the empirical weights as follows, $W_{insert} = 1, W_{delete} = 4, W_{replace} = 4$.

Table 7: Parameters Used in Fine-Tuning LLMs

Supervised Fine-Tuning Parameters:	
Train Batch Size : 4	Learning Rate : 2e-5
Evaluation Batch Size : 4	Weight Decay : 0.00
Learning Rate schedule : Cosine	Warmup Ratio : 0.03
Max Sequence Length: 512	Use Lora: True
In-Context Learning Parameters:	
Max Sequence Length: 512	#Shots : 3

Table 8: The P-Values of VulLibGen (Compared with VulLibMiner)

Approach	Java	JS	Python	Go	Total
Commercial LLMs:					
ChatGPT	2e-13	8e-3	1e-10	3e-1	1e-20
GPT4	1e-18	5e-5	1e-5	3e-5	6e-30
Full SFT on Open-Source LLMs:					
LLaMa-7B	7e-7	1e-5	1e-11	9e-6	1e-25
LLaMa-13B	5e-8	1e-4	1e-9	1e-9	3e-27
Vicuna-7B	5e-5	6e-5	1e-12	5e-10	1e-27
Vicuna-13B	7e-7	1e-5	6e-13	1e-11	4e-32

10.2 Appended Tables

Table 9: Status of Submitted <Vulnerability, Affected Package> Pairs

CVE ID	VulLibGen's Output	Status
CVE-2010-5327	com.liferay.portal:portal-impl	Merged
CVE-2010-5327	com.liferay.portal:portal-service	Merged
CVE-2012-3428	org.jboss.ironjacamar:ironjacamar-jdbc	Merged
CVE-2012-5881	yui2	Merged
CVE-2013-1814	org.apache.rave:rave-web	Merged
CVE-2013-1814	org.apache.rave:rave-portal-resources	Merged
CVE-2013-1814	org.apache.rave:rave-core	Merged
CVE-2014-0095	org.apache.tomcat.embed:tomcat-embed-core	Merged
CVE-2014-0095	org.apache.tomcat:tomcat-coyote	Merged
CVE-2014-1202	com.smartbear.soapui:soapui	Merged
CVE-2014-6071	jquery	Non-Vuln
CVE-2014-9515	com.github.dozermapper:dozer-parent	Non-Vuln
CVE-2015-3158	org.picketlink:picketlink-bindings-parent	Incorrect
CVE-2017-1000397	org.jenkins-ci.main:maven-plugin	Merged
CVE-2017-1000406	org.opendaylight.integration:distribution-karaf	Merged
CVE-2017-3202	com.exadel.flamingo.flex:amf-serializer	Merged
CVE-2017-7662	org.apache.cxf.fediz:fediz-oidc	Merged
CVE-2018-1000057	org.jenkins-ci.plugins:credentials-binding	Merged
CVE-2018-1000191	com.synopsys.integration:synopsys-detect	Merged
CVE-2018-1229	org.springframework.batch:spring-batch-admin-manager	Merged
CVE-2018-1256	io.pivotal.spring.cloud:spring-cloud-sso-connector	Merged
CVE-2018-3824	org.elasticsearch:elasticsearch	Merged
CVE-2018-5653	wordpress/weblizar-pinterest-feeds	Incorrect
CVE-2018-7747	calderajs/forms	Incorrect
CVE-2019-10475	org.jenkins-ci.plugins:build-metrics	Merged
CVE-2019-5312	com.github.binarywang:weixin-java-common	Merged
CVE-2020-8920	com.google.gerrit:gerrit-plugin-api	Merged
CVE-2022-25517	com.baomidou:mybatis-plus	Non-Vuln

"Merged": Its corresponding package name is accepted and merged into GitHub Advisory.
"Non-Vuln": GitHub Advisory's maintainers do not consider it as a vulnerability.
'Incorrect": VulLibGen's output is incorrect and not accepted by maintainers.