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

**Anonymous ACL submission**

## Abstract

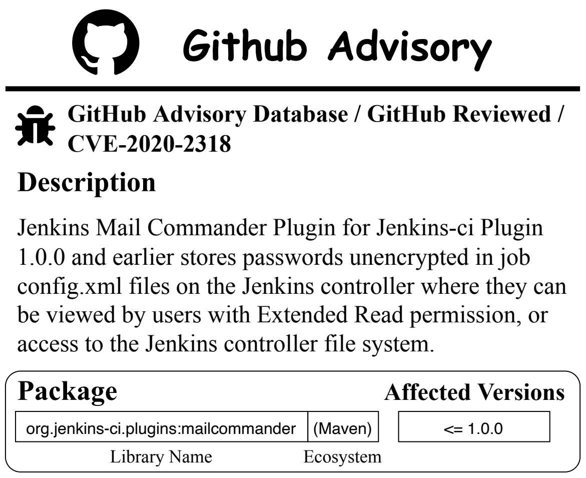
Security practitioners maintain vulnerability reports (e.g., GitHub Advisory) to help developers mitigate security risks. An important task for these databases is automatically extracting structured information mentioned in the report, e.g., the affected software packages, to accelerate the defense of the vulnerability ecosystem.

However, it is challenging for existing work on affected package identification to achieve a high accuracy. One reason is that all existing work focuses on relatively smaller models, thus they cannot harness the knowledge and semantic capabilities of large language models.

To address this limitation, we propose VulLibGen, the first method to use LLM for affected package identification. In contrast to existing work, VulLibGen proposes the novel idea to directly generate the affected package. To improve the accuracy, VulLibGen employs supervised fine-tuning (SFT), retrieval augmented generation (RAG) and a local search algorithm. The local search algorithm is a novel post-processing algorithm we introduce for reducing the hallucination of the generated packages. 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 average accuracy in previous work is 0.721. Additionally, VulLibGen has high value to security practice: we submitted 60 <vulnerability, affected package> pairs to GitHub Advisory (covers four ecosystems). 22 of them have been accepted and merged and 32 are pending approval. Our code and dataset can be found in the attachments.

## 1 Introduction

One important task in security is to automatically extract the package names and versions affected by each security vulnerability. A recent study (Wang et al., 2020) shows that 84% third-party packages contain security vulnerabilities and 60% of them are high-risk ones. To mitigate the security risks, security practitioners maintain databases (e.g., NVD and GitHub Advisory) for each unique vulnerability (i.e., Common Weakness Enumeration or CVE). These databases contain reports with the vulnerability description, affected packages, and affected versions. For example, Figure 1 shows the report of CVE-2020-2318. By linking the CVEs to the affected packages, developers can be aware of the vulnerable packages in their code and quickly apply patches/fixes; the affected packages also help with security knowledge management and task prioritization. Although developers can manually enter the affected packages when submitting CVEs, the entered information is often missing or incorrect (Haryono et al., 2022; Dong et al., 2019). While database maintainers also conduct manual reviews (GitHub, 2024b), automatic identification can accelerate the vulnerability lifecycle and reduce the manual costs, thus it is a critical task.


Github Advisory

GitHub Advisory Database / GitHub Reviewed / CVE-2020-2318

**Description**

Jenkins Mail Commander Plugin for Jenkins-ci Plugin 1.0.0 and earlier stores passwords unencrypted in job config.xml files on the Jenkins controller where they can be viewed by users with Extended Read permission, or access to the Jenkins controller file system.

| Package | | Affected Versions |
|---|---|---|
| org.jenkins-ci.plugins:mailcommander | (Maven) | <= 1.0.0 |
| Library Name | Ecosystem | |

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

Several existing works have studied automatic affected package identification (Chen et al., 2020; Dong et al., 2019; Haryono et al., 2022; Lyu et al., 2023; Chen et al., 2023b), however, it is challenging for them to achieve high accuracy because they fail to leverage large language models. Existing works typically rank/retrieve the package name from a list of pre-defined packages by computing the similarity between the vulnerability description and the package description. Since the time complexity of retrieval is linear to the size of the list (e.g., 435k Java packages), the cost of each model inference has to be kept quite low and only smaller models have been used, e.g., BERT and linear regression (Chen et al., 2020; Lyu et al., 2023; Haryono et al., 2022; Chen et al., 2023b).

To leverage the extensive knowledge and semantic capabilities of LLMs, we propose a different strategy: *generate rather than retrieve* the affected package. Our framework, VulLibGen, is the first framework to directly generate the mentioned package name given the vulnerability description. Since generation requires the LLM inference to be invoked only once, our approach can easily scale to larger language models such as Llama-13B and Vicuna-13B on a single GPU server.

Upon observing errors in the raw generation result of LLMs, we propose the following techniques for improving VulLibGen's accuracy. First, we conduct supervised fine-tuning (SFT), in-context learning, and retrieval-augmented generation (RAG) to enhance the domain knowledge of a LLM. Second, we propose a novel local search technique to post-process the raw output for reducing hallucination, i.e., ensuring that the generated package name actually exists. Our local search algorithm is inspired by an empirical study of the incorrect raw outputs of ChatGPT. The study shows that most errors are partially correct, therefore we can create rules to match the incorrect part to the closest existing one given the correct part. Since the sub-level package information (e.g., `mailcommander` in Figure 1) is often more directly mentioned than the top-level package information (e.g., `org.jenkins-ci.plugins`), our local search algorithm first matches the suffix before the prefix.

We evaluate VulLibGen on four vulnerability ecosystems: Java, JS, Python and Go. Our evaluation attains three main findings. First, we observe that the accuracy of VulLibGen (0.806) significantly outperforms existing ranking approaches using smaller models (Chen et al., 2020; Haryono et al., 2022; Lyu et al., 2023; Chen et al., 2023b) (0.721) and the computational time costs are comparable. Second, our ablation studies show that SFT, RAG, and local search all help improve the accuracy of VulLibGen and SFT contributes to the most improvement. In particular, the fine-tuned open-source Vicuna-13B outperforms the unfine-tuned commercial ChatGPT and GPT-4 models. Our local search algorithm can significantly reduce the hallucination in the original LLM output, and it is especially helpful for longer package names such as Java and Go. Third, VulLibGen provides high value to security practice: at the time of the writing, we have submitted 60 pairs of <vulnerability, affected package> to GitHub Advisory (28 Java, 11 JS, 11 Python, 10 Go). 22 of Java are accepted and merged. JS/Python/Go are pending approval. Additionally, the source code and dataset of VulLibGen is available on our anonymous website [1].

## 2 Existing Work on Vulnerable Package Identification

This section summarizes existing work on affected package identification and analyze the scalability challenge.

**Formal Definition of Affected Package Identification**. Given a security vulnerability (CVE) submitted to a software ecosystem (e.g., GitHub Advisory), the goal of affected package identification is to link the description $q$ of the CVE to an existing software package name $p$ (e.g., a Maven or PyPi package) that is affected by the CVE. An example of the linked package can be found in Figure 1, where the description mentions the affected package `org.jenkins-ci.plugins:mailcommander`.

**Smaller Models Have Lower Accuracies**. Existing approaches on vulnerable package identification (Chen et al., 2020; Dong et al., 2019; Haryono et al., 2022; Lyu et al., 2023) all suffer from lower accuracies (Lyu et al., 2023; Chen et al., 2023b). Given the vulnerability $q$, existing works rank all packages $p$ of the ecosystem by computing the similarity score between the descriptions of $q$ and $p$. Due to the large number of packages (e.g., Maven has 435k packages), existing work cannot afford using large language model to compute the score. All of them thus rely on smaller models, e.g., logistic regression (Gupta et al., 2021; Lyu et al., 2023) and BERT (Haryono et al., 2022; Chen et al., 2023b). Despite various methods introduced for improving the accuracy (Dong et al., 2019; Anwar et al., 2021), the accuracies remain low (Chen et al., 2023b).

[1]https://github.com/anonymous4ACL24/submission1129

Table 1: An Empirical Study on ChatGPT's Incorrect Response in Maven (Java ecosystem)

| 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." | |

**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., 2023b). More specifically, they first use TF-IDF to rank all packages in the ecosystem (435k in Java and 506k in Python), then re-rank the top-512 packages using BERT. The re-ranking approach achieves a reasonable accuracy with a saved inference cost, but there remains a large room for improving the accuracy (Chen et al., 2023b).

## 3 Two Challenges with LLM Generation

In contrast to existing work, we propose the first work, VulLibGen, that leverages LLMs for affected package identification. Due to the scalability challenge of the retrieval approach, our approach directly *generates rather than retrieves* the affected package. VulLibGen thus only need to invoke the LLM inference once for each vulnerability $q$. Nevertheless, there exist two challenges with the generative approach.

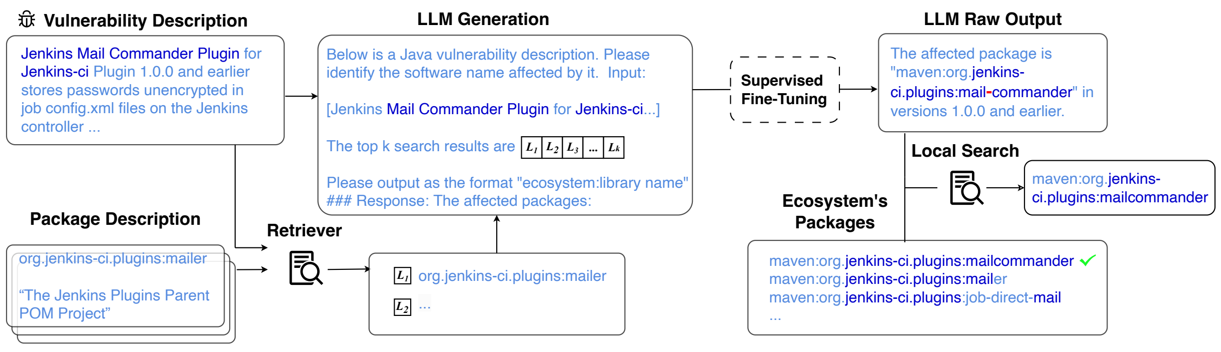
**Challenge 1: Lack of Domain Knowledge.** The first challenge is that there may exist a knowledge gap for the LLM to generate the correct package. This is because the description may not contain the full information about the affected package name. For example, CVE-2020-2167 in Table 1 is about the 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., 2021; Cai et al., 2022).

**Challenge 2: Generating Non-Existing Package Names.** Following a previous study on Reddit[2], the second challenge is that the LLM may generate library names that do not exist in the ecosystem. Existing work has adopted post-processing to reduce the non-existing package issue in code translation and program repair (Jin et al., 2023; Roziere et al., 2022). Following existing work, we can potentially leverage post-processing by matching the generated package with the closest existing package.

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., 2023b). We divide all ChatGPT responses into four types: 1. the package is incorrect but it exists (13%, 23% of errors); 2. the package does not exist and is partially correct (34%, 58% of errors); 3. the package is completely incorrect (11%, 19% of errors). 4. the package is correct (42% of all cases);

From the study result, we draw the conclusion that post-processing by matching is a promising approach to solve Challenge 2. This is because the majority errors are Type 2 errors, while post-processing is the most effective in helping with Type 2 errors. For example, for CVE-2020-2167, ChatGPT generates org.jenkins-ci.plugins:openshift-pipeline. While the suffix is

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



Vulnerability Description
Jenkins Mail Commander Plugin for Jenkins-ci Plugin 1.0.0 and earlier stores passwords unencrypted in job config.xml files on the Jenkins controller ...
LLM Generation
Below is a Java vulnerability description. Please identify the software name affected by it. Input:
[Jenkins Mail Commander Plugin for Jenkins-ci...]
The top k search results are $L_1$ $L_2$ $L_3$ ... $L_k$
Please output as the format "ecosystem:library name"
### Response: The affected packages:
Supervised Fine-Tuning
LLM Raw Output
The affected package is "maven:org.jenkins-ci.plugins:mail-commander" in versions 1.0.0 and earlier.
Local Search
maven:org.jenkins-ci.plugins:mailcommander
Ecosystem's Packages
maven:org.jenkins-ci.plugins:mailcommander ✓
maven:org.jenkins-ci.plugins:mailer
maven:org.jenkins-ci.plugins:job-direct-mail
...
Package Description
org.jenkins-ci.plugins:mailer
"The Jenkins Plugins Parent POM Project"
Retriever
$L_1$ org.jenkins-ci.plugins:mailer
$L_2$ ...

Figure 2: The VulLibGen Framework

correct, the prefix and suffix never co-occur in any existing package name. We can fix this case by matching the prefix to the closest co-occured one. 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 and 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 [3].

### 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 [3], Output: "*The affected package is [package name]*". The hyper-parameters used for ICL and SFT are listed in Table 10 of Appendix.

### 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 existing package names in an ecosystem (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 [4], NPM [5], Pypi [6], and Go [7] 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., 2023b)'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.,

[3]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:*".

[4]`https://mvnrepository.com`
[5]`https://www.npmjs.com`
[6]`https://pypi.org`
[7]`https://pkg.go.dev`

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 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 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, 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**. Among existing work on affected package identification (Chen et al., 2020; Haryono et al., 2022; Lyu et al., 2023; Chen et al., 2023b), two datasets are frequently used: VeraCode (Chen et al., 2020) and VulLib (Chen et al., 2023b). In this work, we choose to use VulLib instead of VeraCode because it is of better quality. The VulLib dataset contains 2,789 Java vulnerabilities collected from GitHub Advisory. Each package name in VulLib is manually verified by security experts from GitHub Advisory (GitHub, 2024a). In contrast, VeraCode is not verified thus it is prone to errors [8]. Since VulLib only focuses on Java, we further extend it to JS, Python and Go by collecting the data from GitHub Advisory following a similar workflow as VulLib.

[8]VeraCode does not use explicit package names ecosystems. For example, the affected package of CVE-2014-2059 is "org.jenkins-ci.main:jenkins-core" while VeraCode labels it as three packages: "jenkins", "openshift-origin-cartridge-jenkins", and "jenkins-plugin-openshift"

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 |

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 existing ranking approaches, FastXML (Chen et al., 2020), LightXML (Haryono et al., 2022), Chronos (Lyu et al., 2023), and VulLibMiner (Chen et al., 2023b) for comparison. Recent studies (Lyu et al., 2023; Chen et al., 2023b) show that they outperform other approaches, such as Bonsai (Khandagale et al., 2020) and ExtremeText (Wydmuch et al., 2018).

**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, 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

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** |

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** Following previous work (Chen et al., 2023b, 2020), we use three metrics for evaluating VulLibGen and baselines: Accuracy@1 (i.e., Prec@1), Recall@1, and F1@1.

## 5.2 Evaluation of VulLibGen

In this subsection, 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 Performance: Existing Work vs. VulLibGen**. In Table 3, Table 7 and Table 8 (Appendix), we compare the performance of VulLibGen and baselines. From these tables 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. In Table 11 of Appendix, we further report statistical significance tests (Kim, 2015) between the overall best-performing generative approach (i.e., VulLibGen using Vicuna-13B SFT) and the best-performing existing work (i.e., VulLibMiner (Chen et al., 2023b)). The p-values in all tests are smaller than $1e$-5.

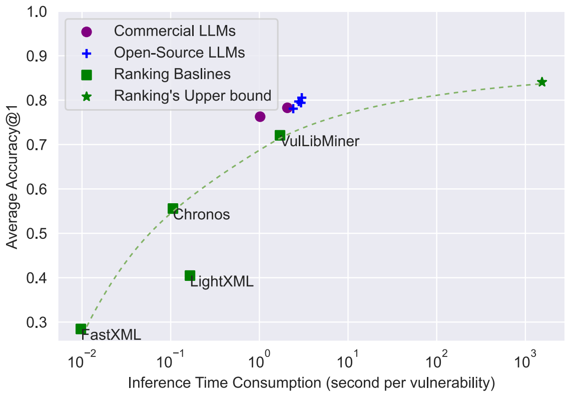


Figure 3: Trade-Offs between Efficiency and Accuracy

**When Is VulLibGen More Advantageous?** From Table 3 observe that the gap between VulLibGen Vicuna-13B SFT 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.

**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 a better Accuracy@1 while the time cost is comparable to the best-performed existing work (Chen et al., 2023b). 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 subsection, we conduct ablation studies on the three components of VulLibGen: supervised fine-tuning, RAG, and local search.

**SFT's Improvement**. By comparing the results

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% ↑ |

Table 5: 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% ↑ |

of in-context learning vs supervised fine-tuning in Table 3, we can see that SFT outperforms ICL by a larger margin. This result indicates that for the 7B and 13B models, supervised fine-tuning on the full training data 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. We further report paired t-test results for Table 4 in Table 13 of Appendix.

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 Choice**. We evaluate whether $k$ and the choice of 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 9 in our Appendix.

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 paired 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 5 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. We further report paired t-test results for Table 5 in Table 12 of Appendix.

We make the following observations. First, local search 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. Second, local search is more effective on Java and Go than JS and Python. The reason is that since Java and Go packages are longer (8-14 tokens), LLMs are more prone to generating partially correct, non-existing outputs (i.e., Type 2 error in Table 1). Local search can effectively reduce this type of error.

## 5.4 Evaluating VulLibGen Performance in Real World Setting

To examine VulLibGen's performance in the real-world setting, for each programming language, we randomly sample and report a subset of <vulnerability, affected package> pairs that are not listed in GitHub Advisory (Java: 28, JS: 10, Python: 10, Go:

10). We use VulLibGen to generate the package names and submit the generated names (VulLibGen with Vicuna-13B) to GitHub Advisory.

At the time of the writing, the results are summarized below. **Java**: 22 of have been accepted and merged into GitHub Advisory. Among the remaining 6 packages, 3 of them are considered non-vulnerabilities, and 3 of them are considered incorrect affected packages. **JS, Python, and Go**: for each programming language, 10 have been submitted and are waiting for the review. The details of these packages are listed in the Appendix (Table 14).

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

## 6 Related Work

**Vulnerable Package/Version Identification**. Numerous existing works have proposed methods to improve the accuracy of affected package identification. Multiple existing works model this problem as a named entity recognition (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 software version identification since many version numbers are already in the description (Dong et al., 2019). On the other hand, the package names are often only partially mentioned (e.g., CVE-2020-2167 in Table 1), therefore the NER approaches are less effective (Lyu et al., 2023). Another 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., 2023b) 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., 2023) 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., 2021; 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-related tasks, existing work (Jin et al., 2023; Chen et al., 2023a; 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 60 pairs of <vulnerability, affected package> to GitHub advisory. 22 of them have been accepted and merged, and 32 are pending approval.

## 8 Limitation

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

**Challenges in Generating Long and Complex Package Names**. As discussed in Section 5.2, the effectiveness of VulLibGen depends on the token length and the number of unique packages. Table 3 shows Java is more challenging than others while having the highest token length and unique packages (Table 2). To improve the generation 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. In particular, it may pose further challenges to generate packages that are exceptionally long. To understand the distribution of token lengths, we report the quantile statistics of the token lengths in Table 6 of Appendix. Table 6 shows that the majority of the package names are shorter than 20 tokens, therefore, exceptionally long package names are very rare.

**Challenges in 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*.

Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in Android and its security applications. In *ACM SIGSAC Conference on Computer and Communications Security*.

Deng Cai, Yan Wang, Lemao Liu, and Shuming Shi. 2022. Recent advances in retrieval-augmented text generation. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023a. CodeT: Code Generation with Generated Tests. In *The International Conference on Learning Representations*.

Tianyu Chen, Lin Li, Bingjie Shan, Guangtai Liang, Ding Li, Qianxiang Wang, and Tao Xie. 2023b. Identifying vulnerable third-party libraries from textual descriptions of vulnerabilities and libraries. *arXiv preprint arXiv:2307.08206*.

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

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*.

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 *USENIX Security Symposium*.

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 *International Conference on Software Engineering*.

Nilesh Gupta, Sakina Bohra, Yashoteja Prabhu, Saurabh Purohit, and Manik Varma. 2021. Generalized zero-shot extreme multi-label learning. In *ACM SIGKDD Conference on Knowledge Discovery & Data Mining*.

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? In *IEEE/ACM International Conference on Program Comprehension*.

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. In *Findings of the Association for Computational Linguistics: EACL*.

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*.

Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

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

Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *IEEE/ACM International Conference on Software Engineering*.

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*.

Tae Kyun Kim. 2015. T test as a parametric statistic. *Korean journal of anesthesiology*.

Philipp Kuehn, Markus Bayer, Marc Wendelborn, and Christian Reuter. 2021. OVANA: An approach to analyze and improve the information quality of vulnerability databases. In *International Conference on Availability, Reliability and Security*.

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*.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jingkai Siow, and Yang Liu. 2021. Retrieval-augmented generation for code summarization via hybrid GNN. In *International Conference on Learning Representations*.

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. In *International Conference on Software Engineering*.

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. In *Advances in Neural Information Processing Systems*.

Yuning Mao, Pengcheng He, Xiaodong Liu, Yelong Shen, Jianfeng Gao, Jiawei Han, and Weizhu Chen. 2020. Generation-augmented retrieval for open-domain question answering. In *Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing*.

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 *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

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*.

Baptiste Roziere, Jie Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging automated unit tests for unsupervised code translation. In *International Conference on Learning Representations*.

Lars St, Svante Wold, et al. 1989. Analysis of variance (ANOVA). *Chemometrics and intelligent laboratory systems*.

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. In *Proceedings of the 19th International Conference on Mining Software Repositories*.

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*.

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 *IEEE International Conference on Software Maintenance and Evolution*.

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 *International Conference on Software Engineering*.

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. In *Advances in neural information processing systems*.

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*.

Wenhao Yu, Dan Iter, Shuohang Wang, Yichong Xu, Mingxuan Ju, Soumya Sanyal, Chenguang Zhu, Michael Zeng, and Meng Jiang. 2023. Generate rather than retrieve: Large language models are strong context generators. In *International Conference on Learning Representations*.

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 *IEEE/ACM International Conference on Software Engineering*.

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

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. In *Annual Meeting of the Association for Computational Linguistics*.

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 *IEEE International Conference on Software Analysis, Evolution and Reengineering*.

## 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  $nameDict, suffixes =\leftarrow \{\}, \emptyset$;
2  for $name \in nameList$ do
3      $prefix, suffix \leftarrow name.split(``/:")$;
4      $nameDict[suffix].add(prefix)$;

// Search the closest prefix/suffix
5  $prefix, suffix \leftarrow rawName.split(``/:")$;
6  $edit.weight \leftarrow (W_{insert}, W_{delete}, W_{replace})$;
7  $suffix' \leftarrow \operatorname{argmin}_{s \in suffixes} edit(suffix, s)$;
8  $prefixes \leftarrow nameDict[suffix']$;
9  if $prefixes.isEmpty()$ then
10     return $\{suffix'\}$;
11 else
12     $prefix' \leftarrow \operatorname{argmin}_{p \in prefixes} edit(prefix, p)$;
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

Table 6: The Quantile Statistics of the Number of Tokens in the Package Names

| quantile | 0.5 | 0.75 | 0.95 | 1 |
|---|---|---|---|---|
| Maven | 13 | 15 | 20 | 40 |
| npm | 4 | 6 | 9 | 16 |
| pypi | 3 | 5 | 8 | 15 |
| go | 13 | 15 | 18 | 34 |

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, $W_{insert} = 1, W_{delete} = 4, W_{replace} = 4$.

## 10.2 The Performance Comparison between VulLibGen and Baselines

The performance comparison (Recall, F1) between VulLibGen and baselines can be found in Table 7 and Table 8.

Table 7: VulLibGen's Recall@1 with Various LLMs

| Approach | Java | JS | Python | Go | Avg. |
|---|---|---|---|---|---|
| *Ranking-based Non-LLMs:* | | | | | |
| Chronos | 0.4 | 0.412 | 0.286 | 0.605 | 0.426 |
| VulLibMiner | 0.520 | 0.709 | 0.499 | 0.544 | 0.568 |
| *Commercial LLMs:* | | | | | |
| ChatGPT | 0.573 | 0.698 | 0.580 | 0.544 | 0.599 |
| GPT4 | **0.596** | 0.714 | 0.542 | 0.580 | 0.608 |
| *Full SFT on Open-Source LLMs:* | | | | | |
| LLaMa-7B | 0.548 | 0.734 | 0.593 | 0.625 | 0.625 |
| LLaMa-13B | 0.554 | 0.73 | 0.601 | 0.656 | 0.635 |
| Vicuna-7B | 0.543 | 0.731 | 0.601 | 0.66 | 0.634 |
| Vicuna-13B | 0.552 | **0.736** | **0.621** | **0.688** | **0.649** |

## 10.3 Statistical Significance Test between VulLibGen and Baselines

**VulLibGen vs the Best Baseline.** Table 11 shows the p-values of VulLibGen between the best-

Table 8: VulLibGen's F1@1 with Various LLMs

| Approach | Java | JS | Python | Go | Avg. |
|---|---|---|---|---|---|
| *Ranking-based Non-LLMs:* | | | | | |
| Chronos | 0.451 | 0.429 | 0.376 | 0.653 | 0.482 |
| VulLibMiner | 0.585 | 0.725 | 0.622 | 0.591 | 0.635 |
| *Commercial LLMs:* | | | | | |
| ChatGPT | 0.653 | 0.715 | 0.710 | 0.591 | 0.671 |
| GPT4 | **0.677** | 0.740 | 0.667 | 0.639 | 0.684 |
| *Full SFT on Open-Source LLMs:* | | | | | |
| LLaMa-7B | 0.619 | 0.753 | 0.722 | 0.667 | 0.694 |
| LLaMa-13B | 0.626 | 0.747 | 0.722 | 0.711 | 0.705 |
| Vicuna-7B | 0.610 | 0.749 | 0.730 | 0.716 | 0.705 |
| Vicuna-13B | 0.621 | **0.754** | **0.746** | **0.741** | **0.719** |

performing generative approach (i.e., VulLibGen using Vicuna-13B SFT) and the best-performing existing work (i.e., VulLibMiner (Chen et al., 2023b)). From this table, we can observe that the p-values of all tests are smaller than $1e$-5, indicating the significant improvement of VulLibGen's effectiveness.

**Significance of Using Local Search.** Table 12 shows the p-values of the paired t-tests for each number in Table 5, i.e., before and after local search. We highlight the p-values that are larger than 0.05 in Table 12. We can observe that local search's improvement is less significant in JS and Python and more significant in Java and Go.

**Significance of Using RAG.** Table 13 shows the p-values of the paired t-tests for each number in Table 4, i.e., before and after RAG. We highlight the p-values that are larger than 0.05 in Table 13. We can observe that RAG significantly improves the effectiveness of VulLibGen in most combinations of LLMs and programming languages.

Table 9: 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 10: 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 11: The P-Values of VulLibGen (Compared with VulLibMiner)

| Approach | Java | JS | Python | Go | Avg. |
|---|---|---|---|---|---|
| *Commercial LLMs:* | | | | | |
| ChatGPT | 2e-13 | 8e-3 | 1e-10 | 3e-1 | 7e-2 |
| GPT4 | 1e-18 | 5e-5 | 1e-5 | 3e-5 | 2e-5 |
| *Full SFT on Open-Source LLMs:* | | | | | |
| LLaMa-7B | 7e-7 | 1e-5 | 1e-11 | 9e-6 | 5e-6 |
| LLaMa-13B | 5e-8 | 1e-4 | 1e-9 | 1e-9 | 4e-5 |
| Vicuna-7B | 5e-5 | 6e-5 | 1e-12 | 5e-10 | 3e-5 |
| Vicuna-13B | 7e-7 | 1e-5 | 6e-13 | 1e-11 | 3e-6 |
| Average | 1e-5 | 1e-3 | 2e-6 | 5e-2 | 1e-2 |

Table 12: The P-Values of VulLibGen Before and After Local Search

| Language | Java | JS | Python | Go | Avg. |
|---|---|---|---|---|---|
| *Commercial LLMs:* | | | | | |
| ChatGPT | 1e-6 | 8e-3 | **0.08** | 1e-8 | 2e-2 |
| GPT4 | 3e-8 | 6e-5 | 2e-4 | 1e-6 | 6e-5 |
| *Fine-Tuned Open-Source LLMs:* | | | | | |
| LLaMa-7B | 3e-5 | 2e-2 | 2e-3 | 9e-6 | 5e-3 |
| LLaMa-13B | 2e-5 | **0.16** | **0.08** | 4e-4 | 6e-2 |
| Vicuna-7B | 2e-6 | 2e-2 | 4e-3 | 2e-3 | 6e-3 |
| Vicuna-13B | 8e-3 | **0.32** | 2e-2 | 8e-3 | 3e-2 |
| Average | 1e-3 | **0.09** | 3e-2 | 1e-3 | 3e-2 |

Table 13: The P-Values of VulLibGen Before and After RAG

| Language | Java | JS | Python | Go | Avg. |
|---|---|---|---|---|---|
| Commercial LLMs | | | | | |
| ChatGPT | 3e-23 | 1e-06 | 3e-49 | 1e-52 | 3e-07 |
| GPT4 | 1e-17 | 2e-02 | 4e-02 | 2e-23 | 2e-02 |
| Fine-Tuned Open-Source LLMs | | | | | |
| LLaMa-7B | 3e-04 | 2e-04 | 3e-04 | 4e-02 | 1e-02 |
| LLaMa-13B | 2e-05 | **0.16** | 5e-03 | 8e-04 | 4e-02 |
| Vicuna-7B | 5e-13 | 8e-03 | 2e-06 | **nan** | 2e-03 |
| Vicuna-13B | 8e-03 | 3e-01 | 2e-02 | 8e-03 | 9e-02 |
| Average | 1e-03 | 8e-02 | 1e-02 | 1e-02 | 3e-02 |

Table 14: 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.