Effective Vulnerable Function Identification based on CVE Description Empowered by Large Language Models

Yulun Wu[*][†]
Huazhong University of Science and Technology
Wuhan, China
alanwu@hust.edu.cn

Ming Wen[*][†][‡]
Huazhong University of Science and Technology
Wuhan, China
mwenaa@hust.edu.cn

Zeliang Yu[*][†]
Huazhong University of Science and Technology
Wuhan, China
yuzeliang@hust.edu.cn

Xiaochen Guo[*][†]
Huazhong University of Science and Technology
Wuhan, China
mouchen@hust.edu.cn

Hai Jin[*][§]
Huazhong University of Science and Technology
Wuhan, China
hjin@hust.edu.cn

Abstract

Open-source software (OSS) has profoundly transformed the software development paradigm by facilitating effortless code reuse. However, in recent years, there has been an alarming increase in disclosed vulnerabilities within OSS, posing significant security risks to downstream users. Therefore, analyzing existing vulnerabilities and precisely assessing their threats to downstream applications become pivotal. Plenty of efforts have been made recently towards this problem, such as vulnerability reachability analysis and vulnerability reproduction. The key to these tasks is identifying the *vulnerable function* (*i.e.*, the function where the root cause of a vulnerability resides). However, public vulnerability datasets (e.g., NVD) rarely include this information as pinpointing the exact vulnerable functions remains to be a longstanding challenge.

Existing methods mainly detect vulnerable functions based on vulnerability patches or Proof-of-Concept (PoC). However, such methods face significant limitations due to data availability and the requirement for extensive manual efforts, thus hindering scalability. To address this issue, we propose a novel approach VFFinder that localizes vulnerable functions based on Common Vulnerabilities and Exposures (CVE) descriptions and the corresponding source code utilizing Large Language Models (LLMs). Specifically, VFFinder adopts a customized in-context learning (ICL) approach based on CVE description patterns to enable LLM to extract key entities. It then performs priority matching with the source code to localize vulnerable functions. We assess the performance of VFFinder on 75 large open-source projects. The results demonstrate that VFFinder surpasses existing baselines significantly. Notably, the Top-1 and MRR metrics have been improved substantially, averaging 4.25X and 2.37X respectively. We also integrate VFFinder with Software Composition Analysis (SCA) tools, and the results show that our tool can reduce the false positive rates of existing SCA tools significantly.

Keywords

Vulnerability Analysis, Vulnerable Function, Large Language Model

ACM Reference Format:
Yulun Wu, Ming Wen, Zeliang Yu, Xiaochen Guo, and Hai Jin. 2024. Effective Vulnerable Function Identification based on CVE Description Empowered by Large Language Models. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695013

[*]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology (HUST), Wuhan, 430074, China
[†]Hubei Engineering Research Center on Big Data Security, Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering, HUST, Wuhan, 430074, China
[‡]Corresponding author. Ming Wen is also affiliated with Wuhan JinYinHu Laboratory.
[§]Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, 430074, China

1 Introduction

Open-source software (OSS) has cast a profound and multifaceted impact on the software development paradigm via enabling effortless code reuse, and the last decade has witnessed a surge towards the number of OSS across various ecosystems. Unfortunately, OSS often contains vulnerability issues, and the amount of unveiled vulnerabilities in OSS has been increasing steadily over recent years [41]. Such issues can pose significant security threats to the entire open-source ecosystem. Therefore, precise and effective assessment towards the threats of disclosed vulnerabilities is significant, and plenty of approaches have been proposed over recent years. Software Composition Analysis (SCA) [50] is a widely used method for assessing vulnerability threats. It matches software dependency versions to identify if they fall within the scope of known vulnerabilities. If so, it alerts users or suggests upgrading to secure versions. Popular SCA tools include OWASP Dependency Check [5], GitHub Dependabot [11], and Snyk [40].

However, the coarse-grained detection strategies as adopted by SCA tools often result in high false positive rates [61]. This occurs because 73.3% of projects declaring the usage of vulnerable libraries do not actually use the vulnerable code logic of the library [61], and therefore are not threatened by the vulnerability. The countless false positives greatly annoy users, overwhelming them with unnecessary alerts. Considering the potential version compatibility issues and other reasons [16, 54], upgrading the vulnerable library may require significant manual efforts, further burdening users in dealing with such alerts. This high false positive rate has become a major pain point for existing SCA tools.

To achieve more precise analysis, a common approach adopted by SCA tools is to perform reachability analysis based on *Vulnerable Function* (VF) [53]. VFs denote those functions associated with the vulnerability, where the vulnerable logic resides that can be exploited by meticulously crafted inputs [23]. Reachability analysis usually examines the call graph of downstream applications to determine if there is a call path from the application to the VF in libraries. It can help to ascertain whether an application is truly threatened by security issues, and thus reduce the false positives that merely rely on coarse-grained library version detection.

Unfortunately, the information of VF is hardly available. The existing vulnerability databases (e.g., the National Vulnerability Database (NVD) [8], GitHub Advisory Database [2], and Snyk Vulnerability Database [9]) do not always provide VFs when disclosing vulnerabilities. To address this issue, there are currently two main methods to localize the corresponding VFs for a given vulnerability: patch-based approach [7, 22, 28] and Proof of Concept (PoC) based approach [23]. *The patch-based approach* considers the function where the content is modified or deleted in the patch commit as the VF. However, this approach presents several limitations. First, obtaining patches is not a straightforward process, with approximately 47.1% of Common Vulnerabilities and Exposures (CVEs) in NVD lacking the corresponding patches [43]. Second, security updates may only constitute a portion of the commit, as developers often tangle security updates with other changes, such as adding features in one commit. This can lead to the identification of spurious VFs, many of which may be false positives. The *PoC-based approach* directly identifies the provided function in the PoC as the VF. This approach is generally more accurate compared to the patch-based approach. However, obtaining PoCs is challenging, usually relying on write-ups from blogs and test files related to the vulnerability, which are often unavailable neither.

Given the importance of identifying VFs and the limitations of existing approaches, we propose a highly adaptable approach called VFFinder that can be generalizable to all CVE vulnerabilities. In particular, VFFinder leverages only the vulnerability descriptions, which are mandatory for each CVE, to retrieve candidate VFs from the vulnerable software. VFFinder utilizes Large Language Models (LLM) combined with In-Context Learning (ICL) to extract key information from vulnerability descriptions, subsequently formulating precise queries to identify vulnerable functions. Specifically, VFFinder consists of four steps: ICL-Enhanced Extractor, Description Corpus Generator, Function Corpus Generator, and Priority Semantic Retrieval. The ICL-enhanced extractor is designed to extract the *attack vector* and *root cause*, where the key information lies, from the description to form an accurate query. This extractor employs an effective pattern embedding algorithm to select demonstration examples for LLM to perform the extraction. VFFinder then processes the extracted attack vector and root cause to generate the description corpus. The Function Corpus Generator processes the source file at the function level to extract local code entities, aiming to create a comprehensive document. To accurately characterize the function, VFFinder further performs invocation augmentation by leveraging the function call graph to enrich the information of the caller and callee functions. Finally, VFFinder utilizes Priority Semantic Retrieval matching to rank the functions and retrieve the Top-K functions as VF candidates for output.

We evaluate the performance of VFFinder on 75 large open-source projects with 77 CVEs. The results demonstrate that VFFinder surpasses existing baselines significantly. Notably, the Top-1 and MRR metrics have been improved substantially, averaging 4.25X and 2.37X respectively. In addition, we conduct ablation studies to analyze the contribution of each component. The results demonstrate that all components significantly contribute to the effectiveness of VFFinder. We also integrate VFFinder with existing SCA tools, where the results show when that our tool can help reduce the false positive rates of SCA tools significantly, which can reduce the burden on developers while ensuring the security of the ecosystem.

In summary, we make the following main contributions:

- **Study:** We investigate which parts of the CVE description are most effective for locating the corresponding VFs. Experiments show that the key entities (*i.e.*, attack vector and root cause) in the description is far more effective in extracting VFs than the other ones. Based on this finding, we propose an IR-based approach for vulnerability function localization.
- **Approach:** We propose VFFinder, an IR-based approach for locating VFs empowered by LLM, which only takes vulnerability descriptions and source code as input. VFFinder leverages ICL to extract key entities from the descriptions to generate corpus, and calculates its relevance score to the function corpus of the project, thereby locating the VFs. To our best knowledge, this is the first application that utilizes LLMs in the task of vulnerablity function identification.
- **Evaluation:** We conduct comprehensive experiments to verify the effectiveness and usefulness of VFFinder. The results indicate that VFFinder not only significantly outperforms the existing baselines but also effectively reduces the false positive rate of SCA, demonstrating its usefulness for downstream tasks.
- **Artifact:** We open-source our tool as well as the datasets at: https://github.com/CGCL-codes/VFFinder. We believe such artifacts can benefit future researches related to vulnerability analysis, especially to the problem of VF identification.

2 Vulnerable Function Identification

In this section, we aim to investigate the challenges that hinder the identification of VF by examining various approaches and their limitations. Generally, two primary methods are commonly employed in identifying the VF of vulnerabilities: patch-based [7, 22, 28] and PoC-based approaches [23]. Patch-based methods involve locating the VF through the analysis of patch commits collected from various sources (e.g., GitHub repository). In addition, PoC-based methods derive the VF by analyzing Proof of Concept (PoC) cases.

2.1 Patch-based VF Identification

Patch-based approaches identify VFs based on the patches of vulnerabilities. Specifically, it considers a function as vulnerable if any part of it is modified or deleted in the commit. Therefore, the effectiveness of this approach depends on the availability of vulnerability patches. However, acquiring vulnerability patches is non-trivial. In addition, the quality of these patches will also significantly affect the effectiveness of patch-based approaches. In summary, this type of approaches mainly face two challenges as follows:

Patch Availability. Locating patches presents a significant challenge to identify the VF at the premise. Xin *et al.* [43] build a crawler based on the rules to identify patches of C/C++ vulnerabilities, while only 66.57% CVEs are covered. Xu *et al.* [57] conduct an empirical study to find that more than half of the vulnerabilities are overlooked in Veracode and Snyk based on 10,070 vulnerabilities across different languages. Furthermore, there also exists around 20% patch inconsistency between the two datasets.

Tangled Patch Commits. It is well-known that a commit may tangle different purposes [15, 26, 32, 49], such as bug fix and new features. Security patches face the same issue. Even if the vulnerability fix commit is not tangled, not all modified functions are directly related to the vulnerability. We apply the aforementioned method to obtain VFs on the patches from the Cross-project Vulnerability Patch Dataset (CVPD) published by Wu *et al.* [53], and find that each patch yields 10.2 VFs on average, and the patch[1] linked to CVE-2020-13949 generated the highest number of VFs, totaling 542. It is obvious that not all these functions are the vulnerable ones.

2.2 PoC-based VF Identification

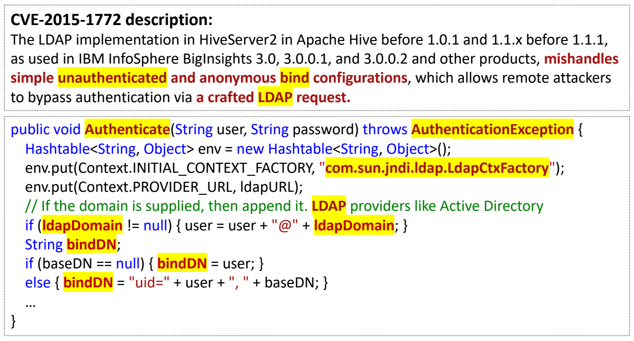
A PoC demonstrates the functionality or viability of a particular idea or method [36]. Under the context of computer security, it is commonly presented in the form of a code snippet, indicating both the VF and the malicious input required to exploit the vulnerability. Essentially, possessing a PoC for a vulnerability means acquiring the details to exploit downstream software dependent on the vulnerable library. Therefore, the VFs identified by these PoCs are more precise compared to patch-based approaches. Unfortunately, such information is also harder to be collected.

There are two main approaches to locate the PoC: explore vulnerability write-ups and inspect the test suite related to the vulnerability patch. Nevertheless, both methods require substantial manual efforts. To assess the effectiveness of these approaches, we examine the top 100 of CVEs based on their CVSS scores in CPVD and performed manual analysis. Our findings reveal that only 24% of the CVEs can obtain the corresponding PoCs in these ways. It is noting that there are datasets of public PoCs provided by ExploitDB[2], and Vulners[3]. However, even including the PoC collected from these sources, this coverage ratio is still only 38%.

3 Motivation and Preliminaries

In this section, we first introduce two concepts closely related to our approach, and then introduce our insights and motivation.

[1]https://github.com/apache/hbase/commit/cb3b06d773a83f007abec55173185402cafa5648
[2]https://www.exploit-db.com/
[3]https://vulners.com/


CVE-2015-1772 description:
The LDAP implementation in HiveServer2 in Apache Hive before 1.0.1 and 1.1.x before 1.1.1, as used in IBM InfoSphere BigInsights 3.0, 3.0.0.1, and 3.0.0.2 and other products, **mishandles simple unauthenticated and anonymous bind configurations**, which allows remote attackers to bypass authentication via **a crafted LDAP request.**

```
public void Authenticate(String user, String password) throws AuthenticationException {
   Hashtable<String, Object> env = new Hashtable<String, Object>();
   env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
   env.put(Context.PROVIDER_URL, ldapURL);
   // If the domain is supplied, then append it. LDAP providers like Active Directory
   if (ldapDomain != null) { user = user + "@" + ldapDomain; }
   String bindDN;
   if (baseDN == null) { bindDN = user; }
   else { bindDN = "uid=" + user + ", " + baseDN; }
   ...
}
```

Figure 1: CVE Description and the Corresponding VF

3.1 Information Retrieval based Approaches

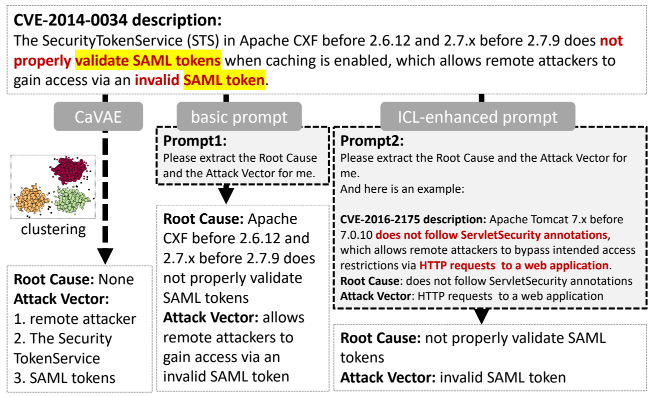
Information Retrieval (IR) [19] is the process of fetching relevant information from a large corpus. IR encodes queries and information of the dataset, and then retrieves knowledge relevant to the query from the dataset by calculating relevance scores. Thus, the essence of IR lies in how the information and queries are represented, and how relevance is computed. IR has been widely applied to tasks such as bug localization in the domain of software engineering [3, 39, 48].

Our proposed approach follows a common IR workflow while using the CVE description as the query and retrieving VF from the vulnerable project. However, utilizing naive IR-based approaches is ineffective in the domain of VF identification since CVE descriptions are often crafted by the CVE submitter according to their own discretion, which are unstructured in nature languages and contain too much irrelevant information. Our key insight to address this issue is that LLMs enable us to extract key entities (*i.e.*, attack vector and root cause) from vulnerability descriptions and critical information from various functions in the corresponding projects (see Section 3.3 for more details). By computing the similarity between those key entities in vulnerability descriptions and function information in the project, we are able to identify the VFs.

3.2 In-Context Learning

In-context learning (ICL) is a paradigm for LLM [14]. Unlike traditional training from scratch, it leverages extensive knowledge and reasoning abilities of the base models. By feeding the input with a few relevant demonstrations, it can guide the model to perform effective reasoning on new tasks. Due to its efficiency and cost-effectiveness, it is widely applied to various downstream tasks [12].

The most crucial part of our approach is leveraging LLMs to extract crucial entities (*i.e.*, attack vectors and root causes) from given descriptions. Specifically, demonstrations in our approach refer to other descriptions with similar patterns to the target CVE. Our insight is that key entities in similar descriptions often appear in similar positions, which can guide the model to better extract these entities. To achieve this, we construct a description pool containing descriptions and their corresponding key entities. We retrieve descriptions from this pool that are similar to the input description and use these descriptions with their key entities as demonstrations to guide the model in entity extraction.



CVE-2014-0034 description:
The SecurityTokenService (STS) in Apache CXF before 2.6.12 and 2.7.x before 2.7.9 does **not properly validate SAML tokens** when caching is enabled, which allows remote attackers to gain access via an **invalid SAML token**.

Root Cause: None
Attack Vector:
1. remote attacker
2. The Security TokenService
3. SAML tokens

Prompt1:
Please extract the Root Cause and the Attack Vector for me.

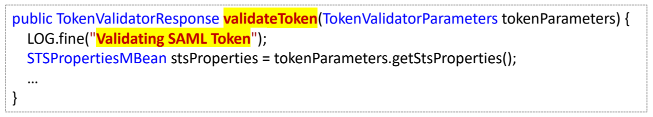
Root Cause: Apache CXF before 2.6.12 and 2.7.x before 2.7.9 does not properly validate SAML tokens
Attack Vector: allows remote attackers to gain access via an invalid SAML token

Prompt2:
Please extract the Root Cause and the Attack Vector for me.
And here is an example:

CVE-2016-2175 description: Apache Tomcat 7.x before 7.0.10 **does not follow ServletSecurity annotations**, which allows remote attackers to bypass intended access restrictions via **HTTP requests to a web application**.
Root Cause: does not follow ServletSecurity annotations
Attack Vector: HTTP requests to a web application

Root Cause: not properly validate SAML tokens
Attack Vector: invalid SAML token

(a) Different Methods of Entities Extraction



```
public TokenValidatorResponse validateToken(TokenValidatorParameters tokenParameters) {
   LOG.fine("Validating SAML Token");
   STSPropertiesMBean stsProperties = tokenParameters.getStsProperties();
   ...
}
```

(b) Vulnerable Function of CVE-2014-0034

Figure 2: A CVE Description and the Extraction of Different Approaches (CaVAE and LLM with Different Prompts)

3.3 Motivating Examples

Both the patch-based and PoC-based VF identification approaches suffer from the data availability problem, thus compromising their scalability across all CVEs. Therefore, to accommodate all CVEs, we need to leverage some generally available information.

We observe that each CVE is associated with the corresponding description, depicting the vulnerability characteristics (e.g., symptom, root cause, and software component). Moreover, some existing research [35, 37, 53] have utilized CVE descriptions to localize VF. However, directly utilizing the description is imprecise for localization, as it can be easily affected by irrelevant information in the descriptions. Eliminating the noise can be challenging because the CVE vulnerability description is often unstructured in nature language, crafted by the CVE submitter according to their own discretion. Therefore, it is hard to extract the key entities from the descriptions for localization.

Fortunately, even though the format of CVE description is in unstructured styles, we observe that there exist certain entities that are required by NVD: vulnerable products, affected version range, and vulnerability type (usually a CWE type). Except for these required entities, several entities are also suggested to be included, such as attack vector, root cause, impact, software component, and attacker. The attack vector defines the method of executing the attack, such as "via a crafted PDF file". The root cause represents a program error or software weakness, like "lack of Item/Read permission". The impact describes the potential consequences of the attack, such as "gaining access to an account without valid credentials". The software component typically refers to a part of the product, such as a file, package, artifact, etc. These optional entities further elaborate the details of the vulnerability. Among these entities, we observe a strong relationship between VFs with attack vectors and root causes, referred to as *indicator entities* in subsequent experiment results.

As shown in Figure 1, in the description of CVE-2015-1772, the attack vector "LDAP request" is similar to the variable names of the VF (e.g., '`ladpDomain`'), and the root cause mentions 'unauthenticated', which is contained in the VF's name, '`Authenticate`'. Besides CVE-2015-1772, CVE-2014-0034 also shows this relationship in Figure 2. The root cause 'validate SAML tokens' closely matches the string literals and name of the VF, '`validateToken`'. The attack vector also includes the terms 'invalid', 'SAML', and 'token', which also share high resemblance to the VF's name and signatures. Such relationships are prevalent among many CVEs such as CVE-2016-0779, CVE-2020-1926, CVE-2021-31409.

Observation I. *Even though the CVE description is unstructured and in various formats, the contained attributes attack vector and root cause are often indicative to help localize vulnerable functions.*

As shown in the above cases, attack vector and root cause are indicative for locating the VF, thus accurately capturing them from the description is crucial. Even though the descriptions are natural language formatted without certain patterns, recent studies reveal that the syntax relation between each entity is relatively fixed. For instance, Yitagesu *et al.* [58] propose a syntax path encoding algorithm CaVAE to capture the syntax relationship between such entities and the vulnerable products, and then utilize unsupervised learning to classify all noun phrases of the description into four categories: attack vector, root cause, impact, and others. However, as such entities are not always described in noun phrase format, CaVAE cannot accurately capture the attack vector and root cause. As shown in Figure 2, the result of CaVAE to extract the entities in not desirable, in which both the root cause and attack vector have not been precisely extracted.

Recently, LLMs have shown excellent capability in various software engineering tasks [6, 13, 27, 42, 60]. However, to the best of our knowledge, no attempts have been made to extract vulnerability entities from descriptions with LLMs. For the above example, we construct a prompt with instructions detailing the task and the entities of attack vector and root cause (*i.e.*, Prompt1 in Figure 2a). We then query the LLM with this prompt. As shown in Figure 2b, the LLM can extract sentences that contain the correct information. Unfortunately, it also includes redundant details such as the product, version, and attacker behavior. Recent studies have shown that ICL [12] can significantly enhance the performance LLMs on various downstream tasks. To help LLMs better recognize the boundaries of each entity and eliminate the noise, we use cases with similar patterns as demonstration examples to format the ICL prompt. These example cases include similar entities like components, products, versions, and attackers, and they follow the same order as CVE-2014-0034. As illustrated in Figure 2b, the answer excludes the vendor, version, and attacker behavior, and directly points out the root cause precisely.

Observation II. *Unsupervised classification is not capable of accurately recognizing the attribute information, while the LLM possesses the capacity to discern it from given descriptions. Furthermore, LLM's performance can be enhanced when it is provided with cases that have a similar structure for ICL.*

4 Approach

Driven by our observations, we present a novel approach for VF localization in this study, named VFFinder (**V**ulnerable **F**unction **Finder**), as depicted in Figure 3. The primary objective of VFFinder is to identify VFs through the analysis of vulnerability descriptions and source code. Its operation aligns with the standard workflow of common IR tasks [19], where a **query** (*i.e.*, the vulnerability description) is used to search for relevant results within a **document** (*i.e.*, the function code corpus set generated from the vulnerable product). ***The key lies in how we represent the query and document via extracting indicative information***.

Figure 3 shows the overview of VFFinder, which contains four main steps, *ICL-Enhanced Extractor*, *Description Corpus Generator*, *Function Corpus Generator*, and *Priority Semantic Retrieval*. The ICL-enhanced extractor is designed to extract the attack vector and root cause from the description, aiming to form an accurate query. Specifically, it employs an effective pattern embedding algorithm to choose demonstration examples for *Large Language Models (LLMs)* to execute the extraction. Subsequently, VFFinder processes the extracted attack vector and root cause to generate the description corpus. The Function Corpus Generator extracts internal code entities by processing the source file at the function level, aiming to create a comprehensive document. To accurately characterize each function, VFFinder further performs invocation augmentation, specifically, leveraging the function call graph to enrich the information of the caller and callee functions. Lastly, VFFinder employs a priority semantic similarity matching to rank the functions, and then retrieve the top K functions as VF candidates for output. The overall design of VFFinder is language-agnostic. It can accommodate VF queries in other programming languages by replacing the corresponding *Function Corpus Generator*.

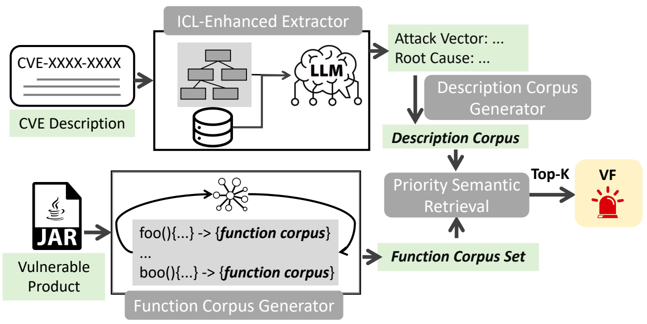


Figure 3: Overview of VFFinder

4.1 ICL-Enhanced Extractor

The ICL-enhanced extractor is employed to extract the attack vector and root cause from the description, serving as the preliminary step in generating the description corpus. It adapts LLMs to perform the extraction tasks through the ICL paradigm without the need for extensive time and resource consumption in fine-tuning. Specifically, we propose a *demonstration retrieval algorithm* and construct a human-labeled case pool to provide high-quality examples for the LLM to learn how to perform effective extraction. It is worth noting that, the extractor works at the sentence level. Therefore, if there are several sentences in one description, the extraction will be conducted for each sentence in the description. The following shows the details.

4.1.1 Case Pool Construction. We construct a case pool to provide high-quality demonstration examples with the attack vector and root cause labeled. Specifically, we first crawl the description that is related to the Java project according to the CVE list provided by CPVD [53], CPVD is the Cross-Project Vulnerablity Dataset published by Wu *et al.*, which consists of 833 CVEs that are related to the Java project, and detailed information about the affected product. In addition, to ensure diversity in the case pool, we randomly select only one CVE when there are consecutive CVE IDs, as the content of these consecutive CVEs is likely to be very similar. Eventually, we retain 669 CVEs and crawl their descriptions.

We then manually label the attack vector and root cause from CVE descriptions. Specifically, we observe that descriptions typically characterize the attack methods with words that express "using" (e.g., with, by, via), followed by a noun phrase indicating the specific method. To minimize noise, we extract only this subsequent noun phrase, which helps us capture the essence of the attack vector without extraneous information. We also observe that some root causes share similar structures as the attack vector (*i.e.*, a noun phrase), typically following phrases such as "due to". We extract this noun phrase as the root cause. Another type of root cause is a verb phrase. The subject prefix of these verb phrases is often the product name, indicating that a specific step was mishandled, leading to the vulnerability. To minimize noise, we only extract this verb phrase as the root cause. Conceptually, there is an overlap between CWE types and root causes. CWE provides a high-level overview of root causes, but its content is too abstract to provide sufficient information for pinpointing VFs. In our annotation process, we deliberately distinguish between CWE and root causes to ensure effectiveness in extraction. It is noted that we label the root cause and attack vector manually. In particular, two authors independently labeled each CVE. We then computed Cohen's Kappa [51] to evaluate the rater agreement. When the two authors disagreed on the labels, they discussed the discrepancies to reach a consensus; otherwise, the CVE was dropped. The entire process took nearly three weeks, resulting in 642 labeled CVEs.

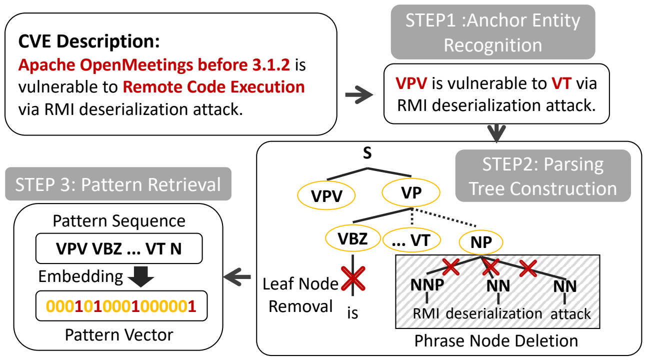

CVE Description:
Apache OpenMeetings before 3.1.2 is vulnerable to Remote Code Execution via RMI deserialization attack.
STEP1 :Anchor Entity Recognition
VPV is vulnerable to VT via RMI deserialization attack.
STEP2: Parsing Tree Construction
S
VPV
VP
VBZ
... VT
NP
NNP
NN
NN
RMI deserialization attack
Leaf Node Removal
is
Phrase Node Deletion
STEP 3: Pattern Retrieval
Pattern Sequence
VPV VBZ ... VT N
Embedding
0001010001000001
Pattern Vector

Figure 4: Demonstration Retrieval

4.1.2 Demonstration Retrieval. To select cases with similar patterns from the pool, we design a customized embedding method for description patterns, which highlights special entities in the descriptions, abstracts grammatical components, and removes irrelevant noise. Figure 4 shows the whole process, which involves the following steps:

Anchor Entity Recognition. The anchor entity refers to four special entities in the description: VPV (*i.e.* vendor, product, and version information), vulnerability type (VT), component, and attacker. The order and distribution of these anchor entities within the description are crucial parts to form patterns. Thus we recognize them and replace them with anchor type (e.g., VPV, VT).

- **VPV**. VPV stands for Vendor, Product, and Version information. We use LLM to identify the VPV entity in a sentence.
- **VT**. The vulnerability type indicates the type of vulnerability, which often appears as CWE-type names in a sentence.
- **Component**. The component refers to the specific affected part of the impacted project. We assist the LLM in identifying the component information by providing the recognized VPV details. If VPV information is not present in a sentence, the component information will not be identified neither.
- **Attacker**. The attacker refers to the entity mentioned in the sentence that performs the attack. We identify this entity by matching the word "attacker".

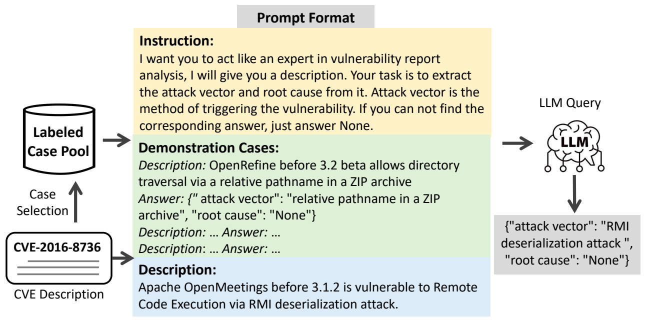
In the example as shown in Figure 4, the "Apache OpenMeetings before 3.1.2" is recognized as VPV, and the "Remote Code Execution" is recognized as vulnerablity type (VT). After identifying the anchor entity in the sentence, we replace the original content with the anchor type. Therefore, the sentence is transformed to "VPV is vulnerable to VT via RMI deserialization attack". These four types of entities are trivial to recognize by LLM or regex match, and they are crucial information in the description, recognizing their positions in the sentences helps highlight the pattern feature.

Parsing Tree Construction. To better capture the relationships between the components within sentences, we use the Stanford parser[4] to construct a parsing tree for the sentences after replacing the anchor entities. All the inner nodes of the tree are grammatical syntactic constructs of phrases, such as sentences (S), noun phrases (NP), verb phrases (VP), etc. All leaf nodes of the tree represent the words and punctuation of the sentence.

Moreover, we use two heuristic methods to prune the parsing tree in order to reduce irrelevant information. First, if a subtree is rooted at a phrase node (e.g., NP or VP nodes) or S node and does not contain other phrase nodes or S nodes, we then delete the child nodes of the subtree, retaining only the root node. As shown in Figure 4, child nodes (*i.e.*, NNP, NN, NN) of the NP node are deleted. The insight is that if a phrase node or S node cannot be further divided, it can be treated as a single compound word. For example, in Figure 4, the "RMI deserialization attack" is viewed as a node, which more clearly reflects the abstract pattern of the description. In addition, we delete all leaf nodes except anchor nodes and phrase nodes. This step further abstracts the sentence's template structure.

Pattern Retrieval. After constructing and refining the parsing tree, we represent each sentence's pattern by concatenating all leaf nodes to form a sequence. We then use Autoencoder [4] to embed each sequence, thus obtaining the pattern vector representation of the sentence. Subsequently, we employ cosine similarity to measure the similarities between pattern embeddings. After calculating the pattern similarity of the queried description against all cases in the pool, we select the Top-K examples with the highest similarities. Such example descriptions, characterized by similar anchor entities and grammatical components, facilitate LLMs to comprehend the semantics and patterns embedded within them.

[4]https://nlp.stanford.edu/software/lex-parser.html



Instruction:
I want you to act like an expert in vulnerability report analysis, I will give you a description. Your task is to extract the attack vector and root cause from it. Attack vector is the method of triggering the vulnerability. If you can not find the corresponding answer, just answer None.

Demonstration Cases:
Description: OpenRefine before 3.2 beta allows directory traversal via a relative pathname in a ZIP archive
Answer: {" attack vector": "relative pathname in a ZIP archive", "root cause": "None"}
Description: ... *Answer:* ...
Description: ... *Answer:* ...

Description:
Apache OpenMeetings before 3.1.2 is vulnerable to Remote Code Execution via RMI deserialization attack.

{"attack vector": "RMI deserialization attack ", "root cause": "None"}

Figure 5: ICL-enhanced Extractor

4.1.3 *Query Design.*

Following previous work [21], we format the prompt as shown in Figure 5 to instruct LLMs to extract attack vector and root cause from given descriptions. It is noted that we choose ChatGPT based on gpt-3.5-turbo-0613[5] to conduct the query. Specifically, the prompt encompasses the following three parts:

- **Instruction**. We briefly introduce the task of A&R (attack vector and root cause) extraction and the concept of A&R in the instruction to enable LLM better understand the task.
- **Demonstration Cases**. Following the instructions, we incorporate three cases with the highest similarity as examples. Each case comprises a description paired with its corresponding A&R. Recent research [14] reveals that LLMs with ICL are particularly susceptible to examples closer to the query. Therefore, we organize the cases in ascending order of similarity to the query. It is based on the intuition that examples with higher similarity may contain more relevant information to the queried description.
- **CVE Description**. We present the description of the input CVE to query the LLM.

4.2 Description Corpus Generation

Based on the A&R obtained by the extractor, VFFinder generates a corpus to represent the description following the steps below:

(i) **Preprocessing.** VFFinder first removes stop words, including standard English stop words and punctuation, and then splits compound words into individual words based on capital letters (e.g., '`validateURL`' is divided into '`validate`' and '`URL`'). Following this, words are converted to their stems using the Porter stemming algorithm [52]. We apply the NLTK [6] to achieve such a goal. This preprocessing ensures that the resulting corpora are concise, facilitating more accurate similarity matching.
(ii) **Class Extraction.** VFFinder identifies class names as a special entity of the description. These class names are identified by applying regular expressions (e.g., `in class: class name`) to the description. If class names are found, they are utilized for class matching in the ranking process (see Section 4.4).

[5]https://platform.openai.com/docs/models/gpt-3-5-turbo
[6]https://www.nltk.org/

(iii) **Weight Calculation.** In the obtained description corpus, some entities appear less frequently compared to others. These entities, such as the 'LDAP' in the attack vector part of Figure 1, are highly unique and often critical for identifying vulnerable functions. Therefore, VFFinder further applies Term Frequency-Inverse Document Frequency (TF-IDF) [44] to compute the **term significance weights** of all the words. The weights are used to calculate the similarity in the ranking module.

4.3 Function Corpus Generation

A function corpus is built to represent each function comprises two parts: the *local corpus*, generated from the code within the function body, and the *interactive corpus*, derived from functions that exhibit call relationships. Recent study has shown that such call relations can provide rich semantic information for a function [47].

The *local corpus* consists of five types of features: the identifier name, function name, parameter name, comments, and string literals. For the former three code entity features, we split the entity into individual words according to camel case notations. For the comments and string literals, we remove punctuation and standard stop words, and split compound words in the same manner as the texts in the CVE descriptions. It is worth noting that the class name is kept intact for each function for class matching.

The *interactive corpus* augments the *local corpus*. Specifically, we use Soot[7] to build call graphs for the project and extensively characterize the function by corpus from the connected function. The interactive corpus consists of two types of features: ***caller parameters*** and ***callee inlines***. In particular, when generating an interactive corpus for a function F, caller parameters refer to the actual parameters passed to F when the invocations are made to F in the project, thus enriching of function input information. The callee inline [18] is a common approach to enrich the information of short methods. For methods within five statements, we union the corpus of the functions called by F as the callee inline of F.

After obtain these two corpora, each function in the project is represented by the union set of the local and interactive corpus.

4.4 Priority Semantic Retrieval

In this step, we rank functions using priority semantic matching. Specifically, we assign a priority rank to functions whose class features match the class indicated by the description. The insight is that once a match is found, the range of potential VF can be significantly narrowed, thereby improving accuracy.

Firstly, we use the embedding model angle-llama-7b-nli-20231027[8], which is based on Llama-2-7b-hf[9] and trained for the Semantic Textual Similarity (STS) task, to generate embedding for entities from both the function corpus and description corpus, and finally utilize the cosine score to measure similarities between the embedding.

The retrieval process is shown in the Algorithm 1, which takes description corpus (d), the list of function corpus ($fSet$), similarity threshold (θ), and K as input, and outputs the Top-K function as the VF candidate. Specifically, to determine the score for a function (line 1 to line 6), VFFinder iterates over all entities in the description corpus (line 3). For each entity, it identifies the corresponding entity in the source code corpus with the highest similarity score (line 4), If this highest score exceeds a predefined threshold (line 5), it indicates a strong relationship between the description entity and the entity in the current function's corpus, and we also call this description entity an ***indicator entity*** in the description.

Algorithm 1: Function Ranking Algorithm

```
Input: description corpus $d$, function corpus set $fSet$, threshold $\theta$, $K$
Output: Top-K functions
1  for $f$ in $fSet$ do
2      $f.score \leftarrow 0$
3      for $entity$ in d do
4          $score \leftarrow$ bestMatch$(entity, f)$
5          if $score > \theta$ then
6              $f.score \leftarrow f.score + score \times entity.weight$
7  $priList, nonPriList \leftarrow$ clzMatch$(d, fSet)$
8  $priList$.sortByScore()
9  $nonPriList$.sortByScore()
10 $sortedFuncList \leftarrow priList + nonPriList$
11 return $sortedFuncList[:K]$
```

VFFinder assigns weights to each entity in the description corpus using TF-IDF that is computed during the description corpus generation phase, which reflects the importance of the entity. Specifically, VFFinder multiplies the highest similarity score by the weight of the description entity and adds this weighted score to the function's overall score (line 6). This scoring process is repeated for each entity in the description corpus, ensuring that the function's score accurately reflects its semantic similarity to the description corpus. After calculating the function score, VFFinder performs class matching among the functions. If a function's class matches the class in the description, the function is added to the priority list; otherwise, to the non-priority list (line 7). VFFinder then sorts both lists by function scores and concatenates the non-priority list to the end of the priority list (line 10). Finally, VFFinder outputs the Top-K functions as the VF candidates.

5 Experiment Setup

We introduce the basic information of the evaluation process for VFFinderin this section, including the process of data collection, the metrics employed, and the organization of the research question.

5.1 Data Collection

To evaluate the effectiveness of the ICL-enhanced Extractor, and performance of VFFinder on VF localization, and its assistance in the downstream tasks, we collect the following three datasets:

A&R Dataset D_{AR}. D_{AR} consists of the labeled attack vector and root cause of each CVE, D_{AR} is used to validate the effectiveness of attack vector and root cause recognition by ICL-enhanced extractor. D_{AR} is the same as the case pool used in the ICL-enhanced extractor.

Vulnerable Function Dataset D_{VF}. In D_{VF}, we collect VF corresponding to each CVE, serving as the ground truth for validating VFFinder. We first gather data from previous work. The exploit generation tool Transfer [24] provides detailed information on the VFs and PoCs of each CVE, so we directly use this dataset as part of our

[7] https://github.com/soot-oss/soot
[8] https://huggingface.co/SeanLee97/angle-llama-7b-nli-20231027
[9] https://huggingface.co/meta-llama/Llama-2-7b-hf

VF dataset. This dataset includes 21 CVEs, with each CVE mapping to one VF. To enlarge the dataset, we first examine the external links provided by the NVD, searching for PoCs. We then follow the PoC-based manual VF identification approach as described in Section 2. If the vulnerability is successfully reproduced as the CVE description, we add it to our dataset. However, the reproduction of CVE is a non-trivial task even manual effort is involved, due to the intricate execution environments and the limited availability of relevant information. Finally, we successfully reproduced 56 CVEs. Combined with the data provided by Transfer, our VF dataset contains a total of 77 vulnerability functions corresponding to 77 CVEs, these CVEs span 30 different CWE types across 75 projects in various domains, including Cryptography, JSON processing, File Compression, etc. Be noted that we do not resort to the patch-based approach since the existing patch commits dataset is often imprecise (see Section 2.1).

Cross-project Vulnerability Dataset D_{CV}. D_{CV} is used to test how effectively the VFs identified by VFFinder can reduce the false positive rate of existing SCA tools. By accessing the publicly available cross-project vulnerability database proposed by Wu *et al.* [53], we gathered downstream software corresponding to the CVEs in D_{VF}. From this database, we collected a total of 62 CVEs, 61 vulnerable software instances, and 9,829 downstream instances.

5.2 Evaluation Metrics

To assess the performance of our proposed VF localization model, we employ three commonly used metrics.

Top@N. which calculates the proportion of vulnerable functions that can be found by inspecting the top N (N =1,2,3,...) entities in the returned suspicious function list. The higher the Top@N value, the fewer the efforts needed by developers to identify the vulnerability, indicating better performance.

MRR. The Mean Reciprocal Rank (MRR) [31] is a statistical measure that takes the average of the reciprocal ranks for a series of queries. In this context, the reciprocal rank of a query is defined as the inverse of the position at which the VF is found. This metric serves as an indicator of the model's effectiveness in identifying the relevant VF. A higher MRR value signifies a superior ranking performance on locating VFs.

Average Entity Coverage and JSI. We use the average entity coverage to measure the effectiveness of the ICL-enhanced extractor. Entity Coverage is calculated as follows: if all entities of the attack vector and root cause are covered, the coverage is 1; otherwise, it is 0. The Average Entity Coverage is the mean coverage across all CVEs. To ensure the extracted content is not redundant, we use the Jaccard Similarity Index (***JSI***) [20], a common metric for comparing the similarity of sample sets. JSI comprehensively measures both the extraction coverage and length. It is calculated as follows: $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$, where A represents the entity set of extracted content and B stands for the ground truth entity set.

5.3 Research Question

We perform extensive experiments to validate the effectiveness of our approach via answering the following research questions.

RQ1: How effective are the attack vector and root cause for VF localization?

This RQ aims to understand the importance of the attack vector and root cause for VF identification. Specifically, we analyze the coverage rate of indicator entities by the attack vector and root cause. A higher coverage rate indicates that the attack vector and root cause have stronger correlations to the VF.

RQ2: How does the ICL-enhanced extractor perform on extracting the attack vector and root cause?

In this RQ, we evaluate the performance of the ICL-enhanced extractor in extracting the attack vector and root cause based on average entity coverage and JSI. Besides, we also measure the length of the extracted entity list. Furthermore, we compare our approach with three baseline models: CaVAE, an LLM without demonstration cases, and an LLM using demonstration cases selected based on semantic similarity. These comparisons help assess the impact of the large language models, ICL, and our proposed pattern embedding towards the extraction of the attack vector and root cause.

Specifically, we introduce the implementation of these tools here. CaVAE is open-sourced for direct use. The LLM without demonstration is only provided with the instruction and description parts of the prompt format, as shown in Figure 5. Semantic similarity demonstration cases are retrieved based on semantic embedding similarity with *angle-llama-7b-nli-20231027* model, which is also utilized in the ranking phase. All LLMs employed in the experiment are ChatGPT (*gpt-3.5-turbo-0613*), with the temperature set to 0 to ensure reproducibility. To prevent the overfitting problem, for LLMs based on ICL, we remove the ground truth corresponding to the query description from the case pool.

RQ3: How does VFFinder perform on VF localization?

We assess the ability of VFFinder on the VF localization task by applying it to D_{VF}. We compare VFFinder with three baseline models. Due to the scarcity of tools specifically for VF localization, we select two models from the API recommendation and bug localization fields, which share the same mechanism of information retrieval between natural language and code:

- **VFDetector**. Similar to VFFinder, VFDetecor [30] locates VFs by analyzing vulnerability descriptions and source code. It uses stemming to extract keywords and employs *tvsm* for similarity matching. Since VFDetector is not publicly available, we meticulously implemented it as described in the paper.
- **FuncVerb**. FuncVerb [56] ranks functions by comparing verb phrases in API names and descriptions with the verb semantic roles in the query. FuncVerb provides a replication package, and we directly use it for the evaluation.
- **DNNLoc**. DNNLoc [25] uses deep learning to rank functions based on the vulnerability description, bug fix history, and source code features. DNNLoc is also publicly available.

RQ4: How do different components contribute to the performance of VFFinder?

VFFinder incorporates a series of components designed to enhance its performance based on the characteristics of vulnerability descriptions. To assess the contribution of these components, we observe changes in the Top-K metrics by removing each corresponding component from VFFinder. Including four parts: (i) ICL-enhanced extractor; (ii) priority semantic retrieval, and two sub-components: (iii) invocation augmentation from Function Corpus Generator;

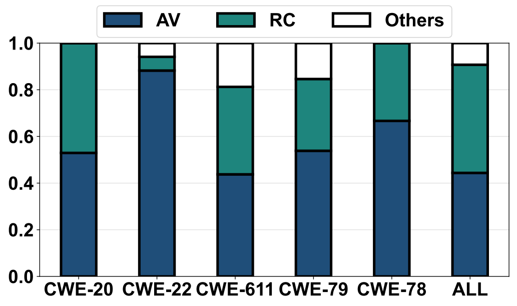


Figure 6: Distribution of the Indicator Entities

and the (iv) description entity weighting from Description Corpus Generator.

RQ5: Can VFFinder benefit the SCA tools?

In this RQ, we aim to utilize the vulnerability functions located by VFFinder to reduce the false positive rate of SCA tools, thereby demonstrating the usefulness of VFFinder. Specifically, we select two SCA tools, Snyk [40] and Dependency Check [5], and construct a cross-project vulnerability dataset D_{CV}. We first obtain the false positive rate of these two tools. Specifically, we first use these two tools to perform software composition analysis on downstream software. For those identified as positive, we use ground truth VF to conduct the reachability analysis. If the VF is not invoked in the downstream software, we consider it a false positive and calculate the false positive rate of the SCA tools. Next, for the positive cases, we use the VFs identified by VFFinder to conduct reachability analysis again. We treat downstream software that does not invoke the identified VF as negative, and ultimately assess the extent to which using these VFs can reduce the false positive rate of the SCA tools.

6 Evaluation result

6.1 RQ1: The Effectiveness of AV and RC

In this research question, we investigate the effectiveness of the attack vector (AV) and the root cause (RC) in localizing VFs based on indicator entity coverage. As indicator entities bridge descriptions and the VFs, higher coverage indicates a greater contribution to accurately pinpointing the VF. Figure 6 illustrates the distribution of the indicator entities across different CWE categories (Top 5 CWEs with most CVEs) and the overall results in all CVEs (ALL). The "others" category refers to the rest part excluding the attack vector and root cause.

Across all CVEs, the attack vector and the root cause together cover 90.8% of the indicator entities, where the attack vector counts for 44.4%, and the root cause slightly higher, 46.3%. The high coverage persists in different CWEs. In CWE-20 and CWE-78, the attack vector and root cause cover 100% of the indicator entities. This substantial coverage implies the attack vector and root cause make a great contribution to the localization. Furthermore, compared to using the full description, we found that when only the attack vector and root cause are used, the average word count in the description corpus drops from 49.4 words to 12.7 words. The removed words include only a few indicator entities, which are not intended to describe the details of the VF but coincidentally match certain code entities. Among these entities, vendor information accounts for the largest proportion (46.2%). Therefore, when these words are removed, the average score of VFs only decreases by 27.5%, while the average score of other non-VFs decreases by 41.2%. This demonstrates that using only the attack vector and root cause can effectively highlight the ranking of VFs.

Answer to RQ1*: The attack vector and root cause encompass most indicator entities in the description, enabling effective VF localization. Utilizing such information instead of the full description reduces the scores of non-VFs while highlighting the VF.*

Table 1: Effectiveness of Vulnerability Entities Extraction

Method	Coverage			Length			JSI		
	AV	RC	AVG	AV	RC	AVG	AV	RC	AVG
CaVAE	0.07	0.07	0.07	39.94	**28.78**	34.36	0.04	0.05	0.04
Direct	0.47	0.35	0.41	41.20	52.45	46.02	0.41	0.29	0.35
S-ICL	0.58	0.55	0.56	31.37	35.71	33.54	0.56	0.53	0.54
P-ICL	**0.60**	**0.58**	**0.59**	**30.49**	35.57	**33.03**	**0.60**	**0.57**	**0.58**

Table 2: Effectiveness of VFFinder and Baselines

Approach	Top@1	Top@3	Top@5	Top@10	Top@30	MRR
VFDetector	4.62%	7.68%	10.77%	18.64%	40.00%	0.078
FuncVerb	0.00%	2.60%	3.90%	6.50%	15.58%	0.022
DNNLoc	6.41%	19.23%	29.49%	35.90%	52.56%	0.166
VFFinder	**27.27%**	**46.75%**	**53.25%**	**63.64%**	**77.92%**	**0.393**

6.2 RQ2: The Effectiveness of the Extractor

As illustrated in settings, we evaluate extractor effectiveness using average entity coverage and check the accuracy and redundancy with JSI. The result is shown in Table 1, the LLM using pattern embedding to obtain cases as examples (P-ICL) demonstrates a higher coverage rate than other models, covering 60% of the attack vector and 58% of the root cause, while achieving a short extraction length of only 33.03 words on average. In terms of the JSI metric, P-ICL reaches 0.58, which is 7% higher than the LLM using cases obtained through semantic embedding (S-ICL). This means that P-ICL can cover more of the attack vector and root cause with shorter output, making it more precise in extracting descriptions.

Compared to the unsupervised classification model CaVAE, the JSI values of the LLM-based model are significantly higher, proving the superior capability of LLM in extracting the attack vector and root cause from complex vulnerability descriptions. P-ICL and S-ICL improve the JSI values by 65.7% and 54.3% respectively, compared to LLM without example cases, indicating that ICL significantly enhances the accuracy of the extraction task.

Answer to RQ2*: LLM can effectively extract the attack vector and root cause from descriptions. Compared to semantically similar examples, using cases with similar patterns helps the LLM better identify attribute boundaries, making the extractor more accurate.*

6.3 RQ3: The Effectiveness of VFFinder

Table 2 compares the VF localization performance of various approaches. VFFinder outperforms the baseline models, showing superior effectiveness. Specifically, it achieves a Top@1 accuracy of 27.27%, which is significantly higher than DNNLoc's 6.41% and VFDetector's 4.62%. while FuncVerb does not successfully identify the correct VF in the first place. For higher K values, VFFinder achieves

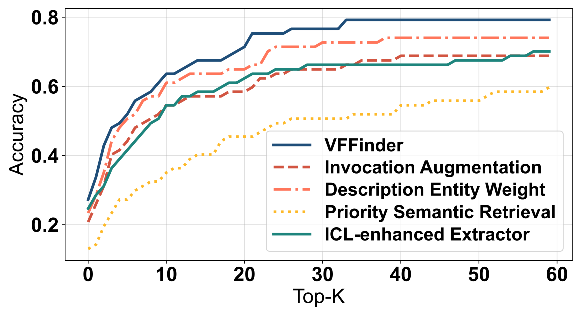


Figure 7: Results of Ablation Study

53.25% for Top@5 and 63.64% for Top@10. The MRR metric for VFFinder is 0.39, which is 2.37 times higher than DNNLoc. Although VFDetector is also designed for VF localization, it performs less effectively because its design inadequately considers the unique features of vulnerabilities. In contrast, VFFinder is designed for these features through vulnerability description analysis, source code representation, and the priority retrieving process.

***Answer to RQ3**: VFFinder shows promising performance in VF localization, achieving a Top-1 accuracy of 27.27%, and MRR reaches 0.39, which is 2.37 times higher than the best baseline methods.*

6.4 RQ4: Ablation Study

Figure 7 illustrates the results of removing different components of VFFinder. Firstly, removing class matching casts the most significant impact on VFFinder, decreasing Top-1 accuracy by 14.2% and an average 25.5% drop, the average drop refers to the mean reduction in accuracy for Top-1 through Top-50 results. This is because class matching prioritizes functions within the same class as mentioned in the vulnerability description, thus narrowing the possible range of VFs and significantly improving accuracy.

The following most impacted components are the ICL-enhanced extractor and invocation augmentation, with an average drop of 10.8% and 10.6%, respectively. These impacts are particularly notable at larger K values, such as Top-20, where the decreases are 9.1% and 13.0%. The ICL-enhanced extractor reduces noise in the retrieval process by accurately extracting indicator entities from the description. Invocation augmentation enhances accuracy by enriching the function corpus with information from inter-function calls. The least impact comes from removing description entity weighting, which results in an average decrease of 4.7%.

***Answer to RQ4**: All components contribute to VFFinder's accuracy, with class matching casting the greatest impact, which significantly narrows the localization range. The following components are the ICL-enhanced extractor and invocation augmentation, which improve accuracy by refining vulnerability descriptions and enriching source function semantics respectively.*

6.5 RQ5: The Usefulness of VFFinder

Figure 8 illustrates the FPR of Snyk and DC, as well as the rates after performing reachability analysis using VF provided by VFFinder. Initially, the average false positive rates of Snyk and DC are 56.1% and 67.1%, respectively. After applying reachability analysis to the top 50 VFs ranked by VFFinder ($VFFinder_{50}$), these rates drop to

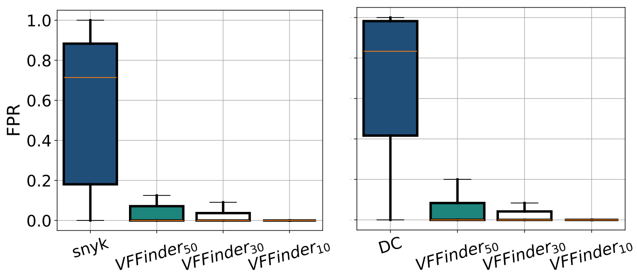


Figure 8: SCA FPR v.s. Reduced FPR by Integrating VFFinder

4.6% and 3.7%. When using the VF identified by $VFFinder_{10}$, the false positive rates of both SCA tools fall below 1%.

To minimize false positives while ensuring no security issues are missed, we recommend SCA tools adopt a tiered alert procedure. For instance, urgent alerts (such as emails or pop-ups) can be sent to developers for issues deemed still threatening after conducting reachability analysis using the VF. Non-urgent alerts (such as sending a GitHub issue) can be used for the remaining. This combination of VFFinder's significant reduction in false positives and a tiered alert system can greatly ease the burden on maintenance personnel while ensuring security of the whole ecosystem.

***Answer to RQ5**: VFs located by VFFinder significantly reduces the FPR of SCA tools. When combined with the results of reachability analysis, employing a tiered alert strategy can alleviate the burden on users caused by SCA.*

7 Related Work

Assessment of Cross-project Vulnerabilities. Given the security risks brought by vulnerabilities of dependencies, i.e., cross-project vulnerabilities, several studies propose methods to assess and mitigate the implications of them to advance existing SCA tools. Wu *et al.* [53] conduct a comprehensive study of cross-project vulnerabilities in Maven ecosystem. Besides assessing reachability from call graph, they also consider the context of callsites and the constraints along the reachable Interprocedural Control Flow Graph (ICFG) path to comprehending potential threats. Targeting bugs of scientific libraries within the Python ecosystem, PyPI, Ma *et al.* [29] first extract conditions that trigger the bugs from bug reports. Subsequently, they evaluate the reachability of the functions with bugs through symbolic execution. Serena *et al.* [33–35] perform studies on upstream vulnerabilities in Maven. They leverage call graph and test suites to assess the reachability of vulnerable functions [38]. Additionally, they integrate fuzzing algorithms to dynamically evaluate the reachability using Evosuite [17]. However, its effectiveness is limited to simple cases, compromising its overall usefulness. Similar to this work, Kang *et al.* [24] further introduce two novel fitness functions of Evosuite, enhancing its ability to guide the fuzzing process toward vulnerable functions. VFFinder is different from prior work in that it provides entry points for evaluating cross-project vulnerabilities and enhances the assessment capabilities of other methods.

LLM for Security. Large Language Models (LLMs) are transforming the field of cybersecurity. These advanced AI models, trained on vast amounts of text data, possess the ability to understand and generate human-like text, making them valuable tools in enhancing security measures. Xia *et al.* [55] propose ChatRepair to generate patches without relying on a vulnerability repair dataset. It aims to improve ChatGPT's code repair capabilities by using a set of successful and failed test cases. They manage to fix 162 out of 337 program vulnerabilities or bugs at a cost of $0.42 per error. TitanFuzz [10] can generate and mutate valid, diverse DL programs to test deep learning (DL) libraries. This approach harnesses the LLMs' understanding of language syntax, semantics, and DL API constraints to enhance fuzzing efficacy. Wang *et al.* [46] introduce a new model called DefectHunter, which uses LLM-driven technology for code vulnerability detection. They demonstrate the potential of combining LLM with advanced mechanisms to more effectively identify software vulnerabilities. In this paper, VFFinder leverages LLMs to refine keywords from CVE descriptions, improving the accuracy of matching vulnerable functions.

Function Retrieval. There are various works to retrieve functions, such as IR-based function-level bug localization, API recommendation. These methods rank and retrieve functions based on their relevance to the given input. Xie *et al.* [56] obtain categories for API methods and propose an API method recommendation approach based on explicit matching of functionality verb phrases in functionality descriptions and user queries. Abid *et al.* [1] combine information retrieval, static code analysis, and data mining techniques to construct the proposed recommendation system FACER. Youm *et al.* [59] present BLIA, which utilizes text and stack traces from bug reports, structured information from source files, and code change history for bug localization. Uneno *et al.* [45] develop DrewBL, constructing a vector space model using semantic-VSM to represent words in bug reports and source code files, and calculating their correlation with word2vec. Mim *et al.* [30] propose an IR-based method called VFDetector for identifying vulnerable functions from source code and vulnerability reports. VFDetector ranks vulnerable functions based on the textual similarity between the vulnerability report corpus and the source code. VFFinder follows the typical IR workflow. However, unlike previous work, it is specially designed for CVE descriptions during document generation and retrieval processes, which helps it better serve the goal of vulnerable function retrieval.

8 Conclusion

In this paper, we present VFFinder, an IR-based approach for vulnerability function localization. This method is the first to leverage Large Language Models (LLMs) for such a task. In particular, VFFinder takes vulnerability descriptions and source function code as inputs, utilizing LLMs with in-context learning to extract key entities from descriptions. VFFinder then encodes these entities as queries and calculates their similarities to entities in the function code, thus locating the vulnerable functions. Our evaluation on 75 large OSS projects shows VFFinder significantly outperforms existing baselines, with Top-1 and MRR metrics improved by 4.25x and 2.37x. Additionally, integrating VFFinder with SCA tools demonstrated an obvious reduction in false positive rates, thereby enhancing the overall effectiveness of these tools. To conclude, VFFinder offers a scalable and effective solution for identifying vulnerable functions in OSS.

9 Acknowledge

We sincerely thank all anonymous reviewers for their valuable feedback and guidance in improving this paper. This work was sponsored by the Major Program (JD) of Hubei Province (No.2023BAA024).

References

[1] Shamsa Abid. 2019. Recommending related functions from API usage-based function clone structures. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 1193–1195. https://doi.org/10.1145/3338906.3342486
[2] GitHub Advisory. 2024. https://github.com/advisories. Accessed: 2024-06.
[3] Shayan A. Akbar and Avinash C. Kak. 2020. A Large-Scale Comparative Evaluation of IR-Based Tools for Bug Localization. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup (Eds.). ACM, 21–31. https://doi.org/10.1145/3379597.3387474
[4] Autoencoder. 2024. https://en.wikipedia.org/wiki/Autoencoder. Accessed:2024-06.
[5] Dependency Check. 2024. https://jeremylong.github.io/DependencyCheck/. Accessed: 2024-06.
[6] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. 2024. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 674–688.
[7] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. 2020. BScout: Direct Whole Patch Presence Test for Java Executables. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1147–1164. https://www.usenix.org/conference/usenixsecurity20/presentation/dai
[8] National Vulnerability Database. 2024. https://nvd.nist.gov/. Accessed: 2024-06.
[9] Snyk Vulnerability Database. 2024. https://security.snyk.io. Accessed: 2024-06.
[10] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 423–435. https://doi.org/10.1145/3597926.3598067
[11] Dependabot. 2024. https://github.com/dependabot. Accessed: 2024-06.
[12] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234* (2022).
[13] Xiaohu Du, Ming Wen, Jiahao Zhu, Zifan Xie, Bin Ji, Huijun Liu, Xuanhua Shi, and Hai Jin. 2024. Generalization-Enhanced Code Vulnerability Detection via Multi-Task Instruction Fine-Tuning. *arXiv preprint arXiv:2406.03718* (2024).
[14] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R. Lyu. 2023. What Makes Good In-Context Demonstrations for Code Intelligence Tasks with LLMs?. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 761–773. https://doi.org/10.1109/ASE56229.2023.00109
[15] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 121–130. https://doi.org/10.1109/MSR.2013.6624018
[16] Kaifeng Huang, Bihuan Chen, Linghao Pan, Shuai Wu, and Xin Peng. 2021. REPFINDER: Finding Replacements for Missing APIs in Library Update. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 266–278. https://doi.org/10.1109/ASE51524.2021.9678905
[17] Emanuele Iannone, Dario Di Nucci, Antonino Sabetta, and Andrea De Lucia. 2021. Toward Automated Exploit Generation for Known Vulnerabilities in Open-Source Libraries. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*. IEEE, 396–400. https://doi.org/10.1109/ICPC52881.2021.00046
[18] Inline. 2024. https://en.wikipedia.org/wiki/Inline_expansion. Accessed: 2024-06.
[19] IR. 2024. https://en.wikipedia.org/wiki/Information_retrieval. Accessed: 2024-06.
[20] Jaccard. 2024. https://en.wikipedia.org/wiki/Jaccard_index. Accessed: 2024-06.
[21] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R. Lyu. 2024. LILAC: Log Parsing using

LLMs with Adaptive Parsing Cache. *Proc. ACM Softw. Eng.* 1, FSE (2024), 137–160.

[22] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 1149–1163. https://doi.org/10.1145/3372297.3417240

[23] Hong Jin Kang, Truong Giang Nguyen, Xuan-Bach Dinh Le, Corina S. Pasareanu, and David Lo. 2022. Test mimicry to assess the exploitability of library vulnerabilities. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*. ACM, 276–288. https://doi.org/10.1145/3533767.3534398

[24] Hong Jin Kang, Truong Giang Nguyen, Xuan-Bach Dinh Le, Corina S. Pasareanu, and David Lo. 2022. Test mimicry to assess the exploitability of library vulnerabilities. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 276–288. https://doi.org/10.1145/3533767.3534398

[25] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, Giuseppe Scanniello, David Lo, and Alexander Serebrenik (Eds.). IEEE Computer Society, 218–229. https://doi.org/10.1109/ICPC.2017.24

[26] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. UTANGO: untangling commits with context-aware, graph-based, code change clustering learning model. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 221–232. https://doi.org/10.1145/3540250.3549171

[27] Bo Lin, Shangwen Wang, Ming Wen, Liqian Chen, and Xiaoguang Mao. 2024. One Size Does Not Fit All: Multi-granularity Patch Generation for Better Automated Program Repair *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1554–1566. https://doi.org/10.1145/3650212.3680381

[28] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. 2020. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1547–1559. https://doi.org/10.1145/3377811.3380923

[29] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yang Feng, Zhaogui Xu, Zhifei Chen, Yuming Zhou, and Baowen Xu. 2020. Impact analysis of cross-project bugs on software ecosystems. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 100–111. https://doi.org/10.1145/3377811.3380442

[30] Rabaya Sultana Mim, Toukir Ahammed, and Kazi Sakib. 2023. Identifying Vulnerable Functions from Source Code using Vulnerability Reports. In *Joint Proceedings of the 5th International Workshop on Experience with SQuaRE series and its Future Direction and the 11th International Workshop on Quantitative Approaches to Software Quality co-located with the 30th Asia Pacific Software Engineering Conference (APSEC 2023), Seoul, South Korea, December 4, 2023 (CEUR Workshop Proceedings, Vol. 3612)*, Tsuyoshi Nakajima, Toshihiro Komiyama, Horst Lichter, Thanwadee Sunetnanta, and Toni Anwar (Eds.). CEUR-WS.org, 66–73. https://ceur-ws.org/Vol-3612/QuASoQ_2023_Paper_04.pdf

[31] MRR. 2024. https://en.wikipedia.org/wiki/Mean_reciprocal_rank. Accessed: 2024-06.

[32] Profir-Petru Pârundefinedachi, Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2020. Flexeme: Untangling Commits Using Lexical Flows. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/3368089.3409693

[33] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable open source dependencies: counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*, Markku Oivo, Daniel Méndez Fernández, and Audris Mockus (Eds.). ACM, 42:1–42:10. https://doi.org/10.1145/3239235.3268920

[34] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2022. Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies. *IEEE Trans. Software Eng.* 48, 5 (2022), 1592–1609. https://doi.org/10.1109/TSE.2020.3025443

[35] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. 2015. Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, Rainer Koschke, Jens Krinke, and Martin P. Robillard (Eds.). IEEE Computer Society, 411–420. https://doi.org/10.1109/ICSM.2015.7332492

[36] PoC. 2024. https://en.wikipedia.org/wiki/Proof_of_concept. Accessed: 2024-06.

[37] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2018. Beyond Metadata: Code-Centric and Usage-Based Analysis of Known Vulnerabilities in Open-Source Software. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 449–460. https://doi.org/10.1109/ICSME.2018.00054

[38] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2018. Beyond Metadata: Code-Centric and Usage-Based Analysis of Known Vulnerabilities in Open-Source Software. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 449–460. https://doi.org/10.1109/ICSME.2018.00054

[39] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 345–355. https://doi.org/10.1109/ASE.2013.6693093

[40] Snyk. 2024. https://snyk.io/product/open-source-security-management/. Accessed: 2024-06.

[41] The state of open source security. 2024. https://snyk.io/reports/open-source-security/. Accessed: 2024-06.

[42] Maolin Sun, Yibiao Yang, Yang Wang, Ming Wen, Haoxiang Jia, and Yuming Zhou. 2023. SMT solver validation empowered by large pre-trained language models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1288–1300. https://doi.org/10.1109/ASE56229.2023.00180

[43] Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. 2021. Locating the Security Patches for Disclosed OSS Vulnerabilities with Vulnerability-Commit Correlation Ranking. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 3282–3299. https://doi.org/10.1145/3460120.3484593

[44] TF-IDF. 2024. https://en.wikipedia.org/wiki/Tf-idf. Accessed: 2024-06.

[45] Yukiya Uneno, Osamu Mizuno, and Eun-Hye Choi. 2016. Using a Distributed Representation of Words in Localizing Relevant Files for Bug Reports. In *2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016, Vienna, Austria, August 1-3, 2016*. IEEE, 183–190. https://doi.org/10.1109/QRS.2016.30

[46] Jin Wang, Zishan Huang, Hengli Liu, Nianyi Yang, and Yinhao Xiao. 2023. DefectHunter: A Novel LLM-Driven Boosted-Conformer-based Code Vulnerability Detection Mechanism. *CoRR* abs/2309.15324 (2023). https://doi.org/10.48550/ARXIV.2309.15324 arXiv:2309.15324

[47] Shangwen Wang, Ming Wen, Bo Lin, and Xiaoguang Mao. 2021. Lightweight global and local contexts guided method name recommendation with prior knowledge. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 741–753. https://doi.org/10.1145/3468264.3468567

[48] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating Bugs from Software Changes. In *Proceedings of the 31st International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 262–273. https://doi.org/10.1145/2970276.2970359

[49] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exploring and Exploiting the Correlations between Bug-inducing and Bug-fixing Commits. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 326–337. https://doi.org/10.1145/3338906.3338962

[50] Wikipedia. 2024. https://en.wikipedia.org/wiki/Software_composition_analysis. Accessed: 2024-06.

[51] Wikipedia. 2024. https://en.wikipedia.org/wiki/Cohen%27s_kappa. Accessed: 2024-06.

[52] Peter Willett. 2006. The Porter stemming algorithm: then and now. *Program* 40, 3 (2006), 219–223. https://doi.org/10.1108/00330330610681295

[53] 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 *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1046–1058. https://doi.org/10.1109/ICSE48619.2023.00095

[54] Laerte Xavier, Aline Brito, André C. Hora, and Marco Túlio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, Martin Pinzger, Gabriele Bavota, and Andrian Marcus (Eds.). IEEE Computer Society, 138–147. https://doi.org/10.1109/SANER.2017.7884616

[55] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *CoRR* abs/2304.00385 (2023). https://doi.org/10.48550/ARXIV.2304.00385 arXiv:2304.00385

[56] Wenkai Xie, Xin Peng, Mingwei Liu, Christoph Treude, Zhenchang Xing, Xiaoxin Zhang, and Wenyun Zhao. 2020. API method recommendation via explicit matching of functionality verb phrases. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1015–1026. https://doi.org/10.1145/3368089.3409731

[57] Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. 2022. Tracking patches for open source software vulnerabilities. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 860–871. https://doi.org/10.1145/3540250.3549125

[58] Sofonias Yitagesu, Zhenchang Xing, Xiaowang Zhang, Zhiyong Feng, Xiaohong Li, and Linyi Han. 2021. Unsupervised Labeling and Extraction of Phrase-based Concepts in Vulnerability Descriptions. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 943–954. https://doi.org/10.1109/ASE51524.2021.9678638

[59] Klaus Changsun Youm, June Ahn, Jeongho Kim, and Eunseok Lee. 2015. Bug Localization Based on Code Change Histories and Bug Reports. In *2015 Asia-Pacific Software Engineering Conference, APSEC 2015, New Delhi, India, December 1-4, 2015*, Jing Sun, Y. Raghu Reddy, Arun Bahulkar, and Anjaneyulu Pasala (Eds.). IEEE Computer Society, 190–197. https://doi.org/10.1109/APSEC.2015.23

[60] Zeliang Yu, Ming Wen, Xiaochen Guo, and Hai Jin. 2024. Maltracker: A Fine-Grained NPM Malware Tracker Copiloted by LLM-Enhanced Dataset. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1759–1771. https://doi.org/10.1145/3650212.3680397

[61] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. 2018. Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 559–563. https://doi.org/10.1109/ICSME.2018.00067