PatchFinder: A Two-Phase Approach to Security Patch Tracing for Disclosed Vulnerabilities in Open-Source Software

Kaixuan Li
East China Normal University
Shanghai, China
Continental-NTU Corporate Lab,
Nanyang Technological University
Singapore, Singapore
kaixuan.li@ntu.edu.sg

Jian Zhang*
Nanyang Technological University
Singapore, Singapore
jian_zhang@ntu.edu.sg

Sen Chen
College of Intelligence and
Computing, Tianjin University
Tianjin, China
senchen@tju.edu.cn

Han Liu
Shanghai Key Laboratory of
Trustworthy Computing, East China
Normal University
Shanghai, China
hanliu@stu.ecnu.edu.cn

Yang Liu
Nanyang Technological University
Singapore, Singapore
yangliu@ntu.edu.sg

Yixiang Chen
Shanghai Key Laboratory of
Trustworthy Computing, East China
Normal University
Shanghai, China
yxchen@sei.ecnu.edu.cn

Abstract

Open-source software (OSS) vulnerabilities are increasingly prevalent, emphasizing the importance of security patches. However, in widely used security platforms like NVD, a substantial number of CVE records still lack trace links to patches. Although rank-based approaches have been proposed for security patch tracing, they heavily rely on handcrafted features in a single-step framework, which limits their effectiveness.

In this paper, we propose PatchFinder, a two-phase framework with end-to-end correlation learning for better-tracing security patches. In the **initial retrieval phase**, we employ a hybrid patch retriever to account for both lexical and semantic matching based on the code changes and the description of a CVE, to narrow down the search space by extracting those commits as candidates that are similar to the CVE descriptions. Afterwards, in the **re-ranking phase**, we design an end-to-end architecture under the supervised fine-tuning paradigm for learning the semantic correlations between CVE descriptions and commits. In this way, we can automatically rank the candidates based on their correlation scores while maintaining low computation overhead. We evaluated our system against 4,789 CVEs from 532 OSS projects. The results are highly promising: PatchFinder achieves a Recall@10 of 80.63% and a Mean Reciprocal Rank (MRR) of 0.7951. Moreover, the Manual Effort@10 required is curtailed to 2.77, marking a 1.94 times improvement over current leading methods. When applying PatchFinder in practice, we initially identified 533 patch commits and submitted them to the official, 482 of which have been confirmed by CVE Numbering Authorities.

CCS Concepts

• **Security and privacy** → **Software security engineering**; • **Information systems** → **Language models**.

Keywords

Security patches, Patch ranking, Large language models

ACM Reference Format:
Kaixuan Li, Jian Zhang, Sen Chen, Han Liu, Yang Liu, and Yixiang Chen. 2024. PatchFinder: A Two-Phase Approach to Security Patch Tracing for Disclosed Vulnerabilities in Open-Source Software. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680305

1 Introduction

Open-source software (OSS) is fundamental to the industrial applications and software community. This widespread adoption, however, comes with security challenges. The open and accessible nature of OSS has inadvertently led to a surge in security vulnerabilities [25, 50, 52]. The notorious vulnerabilities such as Log4Shell [8] and Spring4Shell [6] have put millions at risk of data theft and service denials, reducing trust in the OSS ecosystem.

To facilitate understanding and remediation for vulnerabilities, public security platforms, including the Common Vulnerabilities and Exposures (CVE) and the National Vulnerability Database (NVD), provide details (e.g., descriptions [15]) on disclosed software vulnerabilities and links to relevant patches for mitigation. However, a recent study revealed that almost 57% of CVEs lack trace links to patches, and only 12% of commits in OSS reference the corresponding CVE-IDs [39]. One fact is that the maintainers (e.g., CVE Numbering Authorities, CNAs) may not update the trace link on CVE/NVD even though the vulnerability has been fixed.

*Corresponding author.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680305

This emphasizes the urgent necessity of security patch tracing to enhance the platforms' utility for developers and users.

Yet, pinpointing the exact security patches remains a significant challenge. These patches, which are commits made by OSS developers, are sparsely dispersed throughout individual code repositories. For instance, as of September 2023, the Linux kernel has over 1,215,313 commits, but typically only one of those commits is the patch for a vulnerability. Previous efforts have attempted to search them by leveraging auxiliary information from vulnerability databases, like CVE-IDs in commits [21, 31] and external reference URLs from CVE/NVD pages [48]. However, as evident from the aforementioned statistics, these methods often fall short due to their matching-based nature.

In response to this challenge, the research community has shifted to rank-based methods to pair vulnerabilities with patches. PatchScout [39] and VCMatch [43] are two representative works that build on pair-wise ranking and point-wise ranking respectively. The common practice involves manually defining and extracting features from descriptions and commits, for example, the number of shared words between the description and commit message. Subsequently, the resulting feature vector is used to train a ranking model or classifier. While these rank-based methods offer advantages over traditional matching-based techniques, they also have a set of limitations. First, these methods ***predominantly depend on handcrafted features*** and neglect the rich semantic information. Concretely, ① existing models like PatchScout and VCMatch rely heavily on direct lexical matches (20/22 and 34/36, respectively). The predominantly word-based similarities fall short of capturing semantics including semantic-equivalent phrases and implicit patch information from both sides (CVE descriptions & Commits). ② While two unsupervised features used in existing works provide some benefits, they are not task-specific and cannot fully capture the varying correlations of semantics that are crucial for the task. This limitation can compromise their effectiveness and generality. Second, due to ***the vast gap between the number of developmental commits and security patches***, existing approaches cannot be easily adapted to learn supervised features. The current single-step framework, which involves either directly ranking or classifying commits, presents challenges in effectively training a supervised learning model for the identification of potential patches.

In this paper, we introduce a two-phase framework including initial retrieval and re-ranking, which enables us to learn supervised semantics between commits and CVE descriptions in an end-to-end manner. Our approach, namely PatchFinder, leverages both strengths of Information Retrieval (IR) and Large Language Models (LLMs) to capture lexical and semantic information and commit-related domain knowledge from the two phases respectively. In the initial retrieval phase, we employ a hybrid commit retriever for narrowing down the search space, which consists of TF-IDF (Term Frequency-Inverse Document Frequency) [2] and a pre-trained CodeReviewer. That is, we extract commit candidates that are lexically and semantically similar to the CVE descriptions. In this way, we can sharply narrow the gap as previously discussed. Certainly, this algorithm cannot perfectly understand the correlations. Nevertheless, the refined dataset allows us to mitigate it via supervised learning, which fundamentally differs from existing approaches. In the re-ranking phase, inspired by the success of LLMs on Natural Language Processing (NLP) tasks, we take the CodeReviewer as the foundation model that is tailored specifically for understanding code changes. To unlock its potential, we design an end-to-end architecture under the fine-tuning paradigm [16] for learning the semantic correlations between CVE descriptions and commits. Specifically, we simultaneously encode the description and one commit (including its message and code diff) to obtain two vector representations from the final layer of the CodeReviewer encoder. The vectors are subsequently concatenated into a single vector, which is then fed into a linear classifier to determine whether they are related or not. Together, the hybrid candidate retrieval using TF-IDF and pre-trained CodeReviewer, combined with the semantic correlation captured by fine-tuned CodeReviewer, provides a robust solution to the challenge of tracing security patches.

To comprehensively evaluate our approach, we enrich existing datasets by incorporating newly released CVEs from CVE/NVD, enlarging the dataset from 1,669 to 4,789 CVE entries (4870 patch commits). We then perform extensive experiments on this expanded dataset. The experimental results show that PatchFinder is highly effective and significantly outperforms all baselines. Specifically, PatchFinder boasts a Recall@10 of 80.63% and a Mean Reciprocal Rank (MRR) of 0.7951. Moreover, the Manual Efforts@10 in real-world scenarios are curtailed to just 2.77, marking a notable improvement over the state-of-the-art (SOTA) by 1.94 times. Notably, when deployed in real-world projects, PatchFinder successfully identified 533 new security patches with an average rank of 1.65. Of these, 482 has been confirmed by CNAs.

Our main contributions are as follows.

- We present a two-phase framework for security patch locating: a hybrid initial retrieval phase to refine the search space, followed by a re-ranking phase to learn the correlations between CVE descriptions and patches.
- We design PatchFinder, a comprehensive system that combines the strengths of TF-IDF and CodeReviewer to effectively retrieve potential patch commits while capturing supervised semantics from both descriptions and commits.
- Through extensive evaluations against 4, 789 CVEs from 532 OSS projects, we demonstrate the effectiveness of PatchFinder with a Recall@10 of 80.63% and an MRR of 0.7951, outperforming state-of-the-art methods and reducing manual efforts significantly. Additionally, with the help of PatchFinder, we found and submitted 533 new security patches for the CVE official, of which 482 ones have been confirmed and updated by the CNAs.
- We have released all of our code and data on our website [46] for reproduction and further research.

2 Background and Motivation

2.1 Large Language Models (LLMs)

Pre-trained language models like BERT [7], GPT [33], Llama2 [42], and T5 [34] have significantly advanced NLP tasks. These models adopt a pre-training and then fine-tuning paradigm to develop transferable language representations. This paradigm has been adapted to programming languages with models such as CodeBERT [9], CodeLlama [37], CodeT5 [45], and CodeReviewer [23]. These models have demonstrated remarkable effectiveness, achieving SOTA

performance on various code-related tasks and significantly improving code understanding and generation capabilities.

In specific, CodeReviewer [23] is a pre-trained Transformer-based encoder-decoder language model based on CodeT5 [45]. It was pre-trained with code change and code review data collected from OSS projects on GitHub to support code review tasks for the nine most popular programming languages. Compared with other LLMs, CodeReveiwer has the following characteristics: *1) Purpose-built for Code Change Analysis:* Unlike general-purpose models or those optimized for a broader range of tasks, CodeReviewer is specifically tailored for analyzing code changes. This makes it an apt choice for understanding the semantics of commits, which is pivotal for our task. *2) Pre-training on CodeT5:* CodeReviewer's foundation on CodeT5 means it has benefited from vast amounts of code data during its pre-training phase. This gives it a knowledge advantage over other models that might not have had access to similar training data or might not be as recent as CodeT5.

2.2 Problem Definition

Since it is labor-intensive to trace the security patches for disclosed software vulnerabilities of CVEs, our objective is to design a model that can automatically identify the patches from OSS projects. We view the process as a ranking problem that ranks the commits in OSS projects based on their correlations with reported CVEs. Ideally, when provided with a CVE, the model should rank the associated security patch as high as possible. The model could take the CVE description, commit messages, and code changes (diffs) as input. It then generates a ranked list of commits for the given CVE, indicating the likelihood of each commit being the relevant patch.

Input Data: *CVE Descriptions* and *Commits*. Let $\mathcal{D}$ be the set of CVE descriptions, such that $\mathcal{D} = \{d_1, d_2, \ldots, d_{|\mathcal{D}|}\}$. For each description d_i in $\mathcal{D}$, we have a corresponding set of commits $C_i = \{c_1, c_2, \ldots, c_{|C_i|}\}$. Each commit c_j in C_i is represented as a tuple containing its message and code diff, i.e., $c_j = (msg_j, diff_j)$.

Output Data: *Ranked Commits*. For a given CVE description d_i and its associated commits in C_i, the model produces a ranking vector $\mathbf{R}_i = [r_1, r_2, \ldots, r_{|C_i|}]$. This vector indicates the likelihood of each commit being the patch for d_i, allowing the commits to be sorted based on their rankings.

2.3 Motivating Example

The example in Listing 1 illustrates the challenge of associating a CVE description with its corresponding patch commit [3]. The description for CVE-2015-1867 [30] (Line 2) hints at a vulnerability in *pacemaker* for versions below 1.1.13, but lacks specifics such as its exact location including the function name or file name. Moreover, the commit message (Lines 5-6) provides a hint about the root cause but expresses it differently and does not mention the exploitation of the vulnerability. Existing SOTA tools like PatchScout and VCMatch predominantly rely on commit messages, predefined vulnerability type mappings, and handcrafted features [39, 43]. Such an approach can be limiting, especially when faced with ambiguous CVE descriptions that do not directly match commit messages. Solely relying on token-based features without considering the semantic nuances present in the message and diff can lead to inaccuracies.

```
1  CVE description of CVE-2015-1867:
2  Pacemaker before 1.1.13 does not properly evaluate added nodes,
       which allows remote read-only users to gain privileges via
       an acl command.
3  ****************************************************************
4  Commit message:
5      Fix: acl: Do not delay evaluation of added nodes in some
         situations
6      It is not appropriate when the node has no children as it is
         not a placeholder
7
8  Code diff:
9  diff --git a/lib/common/xml.c b/lib/common/xml.c
10 index f3dd35b7a..716f053f8 100644
11 --- a/lib/common/xml.c
12 +++ b/lib/common/xml.c
13 @@ -1020,13 +1020,16 @@ __xml_acl_post_process(xmlNode * xml)
14 +         char *path = xml_get_path(xml);
15 -         /* Always allow new scaffolding, ie. node with no
16 -            attributes or only an 'id' */
17 +         /* Always allow new scaffolding, ie. node with no
18 +            attributes or only an 'id' Except in the ACLs section
19 +          */
20 -             if (strcmp(prop_name, XML_ATTR_ID) == 0) {
21 +             if (strcmp(prop_name, XML_ATTR_ID) == 0 &&
22 +                 strstr(path, "/"XML_CIB_TAG_ACLS"/") == NULL) {
23 @@ -1035,7 +1038,6 @@ __xml_acl_post_process(xmlNode * xml)
24 -                 char *path = xml_get_path(xml);
25 @@ -1046,6 +1048,7 @@ __xml_acl_post_process(xmlNode * xml)
26 +         free(path);
```

Listing 1: A motivating example for CVE-2015-1867.

In contrast, a more in-depth analysis of the commit, as demonstrated in our motivating example, reveals critical semantic insights that are essential for accurate tracing. Specifically, in Line 14 and Line 26, there is an addition of a new variable `path` which retrieves the path of the XML node. This suggests that the location or context of the XML node in the document might be significant for the fix. It then modifies a comment to specify an exception for the ACLs section (Lines 15-19), indicating that the behavior of allowing new scaffolding nodes is being refined. The condition in Lines 20-22 is enhanced to include a check that ensures the current XML path is not within the ACLs section, as indicated by the string `XML_CIB_TAG_ACLS`. This is a direct response to the vulnerability mentioned in the CVE description, which *allows privileges escalation via an acl command*. The removal of the `path` variable assignment is shown at Line 24, which is now redundant due to its declaration and assignment at the beginning of the block.

It is evident that while the commit message primarily reflects the CVE description, the semantic information in the code diff provides a clearer picture of how the vulnerability is addressed. This underscores the importance of analyzing complex semantics from code diffs in conjunction with commit messages to accurately link CVE descriptions with their corresponding patches.

3 Approach

3.1 Overview

In this section, we present an overview of our approach designed to trace security patches for disclosed vulnerabilities from the NVD/CVE websites. As highlighted in earlier discussions, the primary challenge lies in the implicit correlations between CVE descriptions and commits, which necessarily require understanding the semantics of both sides. Theoretically, we recognized the challenges posed by direct retrieval using a fine-tuned LLM in extensive

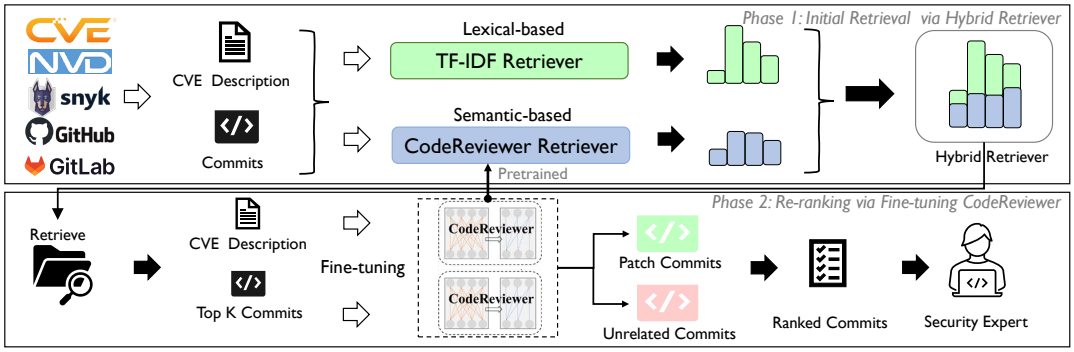

Figure 1: Overview of our approach.

while extremely imbalanced datasets could diffuse the model's attention, reducing its effectiveness in accurately identifying relevant patches. To minimize the training loss, the features of the minority class (i.e. security patches) are easily treated as noise and are often ignored. Thus, there is a high probability of misclassification of the minority class as compared to the majority class (i.e. non-patch commits). Modifying the loss function alone results in a substantial computational burden during the fine-tuning process of LLMs on datasets exceeding 20 million entries.

To tackle these drawbacks, as illustrated in Figure 1, we propose a novel two-phase framework called PatchFinder, comprising initial retrieval and re-ranking. Initially, we retrieve the top candidates of patches from the commits based on lexical and semantic information, which helps eliminate a majority of unrelated commits, such as developmental code changes. On top of that, we design an end-to-end architecture based on LLMs to effectively capture the precise semantics of CVE descriptions and commits. Technically, in the *initial retrieval* phase, we commence by preprocessing the CVE descriptions, commit messages, and code diffs. Subsequently, we employ the TF-IDF and pre-trained CodeReviewer to compute similarity scores between the given CVE description and each code commit at lexical and semantic levels, respectively. Note that this phase is not merely a preliminary step for narrowing down the search space but is critical for ensuring that the re-ranking phase can operate with enhanced focus and accuracy (further details in Section 3.2). Transitioning to the re-ranking phase, we harness the capabilities of LLMs, specifically CodeReviewer, which was pre-trained for code change analysis and defect understanding [23]. By fine-tuning CodeReviewer with the top k commits retrieved in the initial retrieval phase, we aim to deeply comprehend the code semantics present within each code commit. This is particularly crucial for discerning nuances related to security patches (details in Section 3.3).

3.2 Initial Retrieval via Hybrid Retriever

In the vast landscape of open-source repositories, developmental commits overwhelmingly outnumber security patches. To illustrate, the renowned Linux kernel project [41] has amassed 1,215,313 commits as of 15th September 2023. Yet, throughout its history, it has been associated with only 4,165 CVEs [4]. While existing ranking-based methods such as PatchScout [39] have made notable strides, they predominantly lean on handcrafted features to pinpoint security patches. Given the overwhelming number of commits, these methods might not fit well to consistently attain the desired precision. To address this, we incorporate initial retrieval into the security patch tracing task. Specifically, we utilize a hybrid approach to combine a lexical-based TF-IDF [2] retriever and a semantic-based CodeReviewer (pretrained) retriever to take both lexical and semantic information into account. This is because, prior works [19, 49] show that sparse and dense retrievers can complement each other for more robust text retrieval. Due to the existence of large commits and the length constraint of CodeReviewer (maximum of 512 tokens), we preprocess diff files by extracting *only the lines that involve code changes* and then limit the scope to the first 1,000 lines. The statistics show that it can get good coverage (98.6%) of the patch samples on our dataset.

3.2.1 Lexical-based Retriever. TF-IDF [2] stands out for its efficiency and its well-recognized capability to capture lexical similarities. At this stage, our objective is not to definitively locate the security patches but to considerably narrow down the pool of potential commits. By harnessing the capabilities of TF-IDF, we can effectively filter out commits less likely to be security patches, paving the way for a more in-depth analysis in the following stages of our approach.

Formally, in our task, the term t represents individual words or tokens present in CVE descriptions or commits, which includes both commit messages and code diffs. Given a CVE description $d_i \in \mathcal{D}$, both d_i and the corresponding commits $c_j \in C_i$ are treated as separate documents. The entire set of commits associated with a particular CVE, denoted as C_i, forms our corpus for d_i.

The TF-IDF score for a term t in a document d (either a CVE description d_i or a commit c_j) within the corpus C_i is given by:

$$\text{TF-IDF}(t, d, C_i) = \text{TF}(t, d) \times \text{IDF}(t, C_i) \quad (1)$$

Here $\text{TF}(t, d)$ is the term frequency of t in d, calculated as the number of times t appears in d divided by the total number of terms in d. $\text{IDF}(t, C_i)$ is the inverse document frequency of t in C_i, calculated as the logarithm of the total number of documents in C_i divided by the number of documents containing t.

To measure the similarity between the TF-IDF vectors of a given CVE description d_i and a code commit c^j ($c^j \in C_i$), we employ

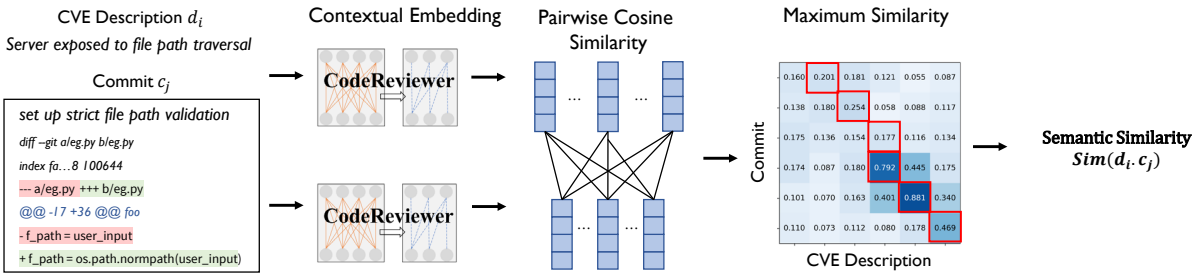

Figure 2: The workflow of our Semantic-based Retriever.

cosine similarity between two vectors $\vec{d_i}$ and $\vec{c_j}$, which is defined as:

$$cosine(\vec{d_i}, \vec{c_j}) = \frac{\vec{d_i} \cdot \vec{c_j}}{\|\vec{d_i}\| \times \|\vec{c_j}\|} \tag{2}$$

where $\vec{d_i} \cdot \vec{c_j}$ is the dot product of the vectors, and $\|\vec{d_i}\|$ and $\|\vec{c_j}\|$ are the magnitudes of the vectors $\vec{d_i}$ and $\vec{c_j}$, respectively. The cosine similarity score ranges between 0 and 1, indicating the lexical similarity between the description and the commit. In this way, commits that are more similar to the given CVE description will have a higher cosine similarity score.

3.2.2 Semantic-based Retriever. Inspired by [51], we adopt a pretrained CodeReviewer model to retrieve relevant patches by measuring their semantic similarity. Specifically, to encode the CVE description and commits, we use a CodeReviewer encoder to map each CVE description and commit pair (d_i, c_j) (where $d_i \in \mathcal{D}$, $c_j \in C_i$) to a fixed-size dense vector, leveraging its proficiency in analyzing code changes and understanding the semantics of defects such as vulnerabilities. Specifically, given a CVE description $d_i =< d_i^1, d_i^2, ..., d_i^{|d_i|} >$ and a candidate commit $c_j =< c_j^1, c_j^2, ..., c_j^{|c_j|} >$, we use contextual embeddings to represent the tokens and compute matching using cosine similarity (as shown in Figure 2).

Token Representation. We use contextual embeddings to represent the tokens in the CVE description d_i and candidate commit c_j, since its better semantic capturing when compared with word embeddings [51]. Contextual embeddings can generate different vector representations for the same word in different sentences depending on the surrounding words, which form the context of the target word. Specifically, We use a shared pretrained CodeReviewer (abbr. CRP) to separately encode the CVE description d_i in $\mathcal{D}$ and each commits candidate c_j in C_i. We prepend a special token of `[CLS]` into its tokenized sequence and employ the final layer hidden state of the `[CLS]` token as the patch representation. We format each of the commits as $\{[CLS], diff_j, [MSG], msg_j\}$. Then the CVE description d_i and each commit $c_j \in C_i$ are fed separately into the CRP encoder to obtain the sequences of token vectors, which can be formulated as: $S(d_i) = CRP_{\text{encoder}}(d_i)$, and $S(c_j) = CRP_{\text{encoder}}(msg_j; diff_j)$ respectively.

Similarity Calculation. The token representation facilitates a soft measure of similarity instead of exact-string or heuristic matching in lexical-based methods. For each token vector in the CVE description, we denote them as $d_i^m \in S(d_i)$ and commit $d_j^n \in S(c_j)$, respectively. Then we calculate their cosine similarity to consider token relations between them. To reduce the calculation cost to the inner product $d_i^{mT} c_j^n$, we use pre-normalized vectors. While this measure considers tokens in isolation, the contextual embeddings contain information from the rest of the sentence.

Based on this, we calculate the complete score that matches each token in CVE description d_i to a token in candidate commit c_j to compute Recall and each token in c_j to a token in d_i to compute Precision. We use greedy matching to maximize the matching similarity score, where each token is matched to the most similar token in the other sentence. Finally, we combine precision and recall to compute an F1 measure. For a CVE description d_i and its candidate commit c_j, the Recall (R), Precision (P), and F1 score (F1) are calculated as:

$$R = \frac{1}{d_i} \sum_{d_i^m \in d_i}^{|d_i|} \max_{c_j^n \in c_j} c_i^{mT} d_j^n \tag{3}$$

$$P = \frac{1}{c_j} \sum_{c_j^n \in c_j}^{|c_j|} \max d_i^{mT} c_j^n \tag{4}$$

$$sim(d_i, c_j) = F1 = 2\frac{R \cdot P}{R + P} \tag{5}$$

The F1 similarity score ranges between 0 and 1, indicating the semantic similarity $sim(d_i, c_j)$ between the given CVE description d_i and the commit c_j. In this way, commits that are more semantically similar to the CVE description will have a higher F1 score.

3.2.3 Hybrid Retriever. To take both lexical and semantic information into account, we utilize a hybrid approach following [19] to combine TF-IDF and CodeReviewer. The fusion of lexical and semantic similarities leverages their complementary analysis perspectives—lexical for word-based similarity and semantic for conceptual alignment (e.g., synonyms). Additionally, they share the same value space ([0,1]), facilitating straightforward additive fusion. The parameter λ adjusts the emphasis on these features, allowing for a unified similarity metric. The similarity score is computed as $f_\phi(d_i, c_j) = sim(d_i, c_j) + \lambda \cdot cosine(\vec{d_i}, \vec{c_j})$, where λ is a weight to balance the two retrievers. After conducting a parameter tuning process including a grid search over various values (from 0.1 to 10 with a step of 0.05), we found that $\lambda = 1$ in our experiment delivers optimal effectiveness among them. Nevertheless, we retain the parameter λ to facilitate adaptation to different datasets.

Based on this combined similarity score, we rank the commits for a given CVE description. We retain the top-k commits that have the highest similarity scores as candidates for the security patch. This yields a refined set of commits, which significantly narrows down the search space for locating the true patch. Indeed, a trade-off exists between efficiency and accuracy in this phrase. While there could be more accurate alternatives for retrieving these candidates such as BM25 [36] and supervised dense retrieval approaches [19, 20], the re-weighting and re-training process adds extra complexity. Fundamentally, we can further analyze the candidates and identify the patch through re-ranking. We provide the details of the re-ranking phrase in the next section.

3.3 Re-ranking via Fine-tuning CodeReviewer

As mentioned above, we have refined the list of commits to the top-k most relevant candidates for each CVE. For this phase, we opt for CodeReviewer [23], a state-of-the-art large language model pre-trained on the foundation of CodeT5 [45]. There are two considerations for this choice. **1) Encoder Specialization:** CodeReviewer's encoder is designed to deeply understand commit behaviors and issues, a feature not necessarily present or optimized in other models. This encoder specialization ensures that the model comprehends the intricate relationships between code changes and potential security implications, vital for matching commits to CVE descriptions. **2) Downstream Task Optimization:** Although our focus is not on generating code reviews, the fact that CodeReviewer's decoder is optimized for such downstream tasks indicates its ability to link code changes to descriptive text, a parallel to our objective of linking commits to CVE descriptions. Given these advantages, we fine-tune CodeReviewer on the top-k candidate commits, aiming to re-rank them based on their relevance to the respective CVE descriptions.

We only use the pre-trained encoder of CodeReviewer (abbr. CR) since our task can be basically viewed as a binary classification problem in the re-ranking phrase. Specifically, given a CVE description d_i, and the top-k commits represented as $C_k = \{(msg_j, diff_j)\}_{j=1}^{k}$, we format each commit as $\{[CLS], diff_j, [MSG], msg_j\}$. Then the CVE description d and each commit $c_j \in C_k$ are encoded separately using the CR encoder to yield two sequences of vectors:

$$E(d) = CR_{\text{encoder}}(d) \tag{6}$$

$$E(c_j) = CR_{\text{encoder}}(msg_j; diff_j) \tag{7}$$

We obtain the vector representations of d_i and c_j by extracting the hidden state in the last layer of the special token `[CLS]` at the beginning of $E(d_i)$ and $E(c_j)$ respectively. The encoded vectors of the CVE description and the commit are concatenated:

$$V_j = [E(d_i); E(c_j)] \tag{8}$$

We apply a linear classifier to the concatenated vector V_j for estimating the correlations:

$$y_j = \sigma(W \cdot V_j + b) \tag{9}$$

where W is the weight matrix, b is the bias term, and σ denotes the sigmoid function ensuring the output lies between 0 and 1.

We utilize labeled data containing known CVE-commit pairs during the fine-tuning phase. The training goal is to minimize the binary cross-entropy loss:

$$\mathcal{L} = -\frac{1}{k}\sum_{j=1}^{k}[y_{\text{true},j}\log(y_j) + (1 - y_{\text{true},j})\log(1 - y_j)] \tag{10}$$

where $y_{\text{true},j}$ is the ground truth label of the j^{th} sample, indicating whether commit c_j is related to the CVE description d_i.

After fine-tuning, for any new CVE and set of commits, the model can compute the relevance scores. Commits can then be re-ranked based on the scores concerning the given CVE. This method ensures that the CR encoder understands both generic textual semantics and the specific indicators that tie a commit to the given CVE, making the approach specially tailored for our challenge.

4 Evaluation

4.1 Research Questions

We aim to answer the following research questions:

- **RQ1: Effectiveness Analysis.** How effective is PatchFinder when compared with existing baselines in tracing security patches?
- **RQ2: Ablation Analysis.** What impact does each component of PatchFinder have on the overall performance?
- **RQ3: Distribution Analysis.** Does PatchFinder exhibit notably high or low accuracy for certain vulnerability types or severity?
- **RQ4: Practicality Analysis.** How effective is PatchFinder in real-world applications, particularly when detecting security patches for CVEs without associated trace links?

4.2 Dataset

For the training and evaluation of our model, we compiled a dataset that encompasses both OSS vulnerabilities and their corresponding security patches. The dataset was assembled in two primary steps: **1) Initial Data Collection:** We began by collecting data from Wang et al. [43] and Tan et al. [39]. We thereby obtained 1,669 unique CVEs from 10 OSS projects. **2) Dataset Supplement:** To augment our dataset, we crawled vulnerabilities and their associated patches from multiple sources, including the official CVE, NVD, and Snyk [38] vulnerability databases. This initial collection yielded 3,585 unique CVEs from 532 OSS projects. After eliminating duplicate entries, our finalized dataset comprises 4,789 unique CVEs and 4,870 distinct security patch commits, which involve 532 unique OSS projects, covering various programming languages including C/C++, Java, JavaScript, and PHP. This makes our dataset the most extensive collection of CVEs and security patches available to date. Each dataset entry includes the CVE-ID along with its textual description and the corresponding security patch links. We also extracted related commits including commit messages and code diffs, primarily from GitHub [11] and GitLab [12].

To create a robust training set, we followed the practices in prior works [39, 43] to sample 5,000 non-patch commits as negative samples for each CVE. However, in scenarios where a repository contained fewer than 5,000 commits in total, we included all available non-patch commits as negative samples. Finally, we got **21,781,044** commits in total. To the best of our knowledge, this is the largest dataset specific for the security patch tracing problem.

4.3 Experiment Setup

We randomly split the 4,789 unique CVEs along with their corresponding commits in the proportion of 8 : 1 : 1 to keep the same split settings as baselines [39, 43]. The maximum token length for CodeReviewer is set at 512, which represents its upper limit for processing capacity. Given the preprocessing of code diffs as detailed in Section 3, this token length is actually sufficient for our purposes. To preprocess data, we use the NLTK toolkit and the BPE tokenizer of CodeReviewer to tokenize CVE descriptions and commits. For the initial retrieval phase, we retrieve the top 100 commits from the initial 5,000 commits for each CVE. This threshold k enables us to obtain a good balance between the coverage of true patches and noisy commits. During fine-tuning, the batch size is set to 32 and the maximum epoch is 20. We adopt the widely-used Adam [22] as the optimizer with a learning rate of 5e-5 for training our model. All the above hyper-parameters are determined based on the validation set by selecting the best ones among some alternatives. All experiments ran on a server with 48 CPU cores (Intel® Xeon® Silver 4214 CPU @ 2.20GHz), 252 GB RAM, and 4 NVIDIA RTX 3090 GPUs (24 GB memory each).

4.4 Baselines

We benchmark our approach against state-of-the-art works in security patch tracing as presented in [39, 43]. Due to the absence of available replication packages of PatchScout [39], we implemented it independently by adhering to the default settings unless specified otherwise. Additionally, we consider two renowned baselines frequently used in the information retrieval domain for our evaluation: BM25 [36], a classic method for sparse retrieval [32], and ColBERT [20], known for dense retrieval [19].

4.5 Evaluation Metrics

To ensure a fair comparison between PatchFinder and the baselines, we utilize three metrics: Recall@K, Mean Reciprocal Rank (MRR) [14], and Manual Efforts@K. Recall@K and Manual Efforts@K have been previously employed in previous studies [39, 43]. We also incorporate MRR into our evaluation, given its established significance in ranking systems.

4.5.1 Recall@K. Recall@K refers to the ratio of the number of security patches traced in the top-K results to the number of all security patches for a given K. Hence, a higher Recall@K score means better performance.

4.5.2 Mean Reciprocal Rank (MRR). MRR is a widely used evaluation metric for ranking systems, particularly in the domain of information retrieval. It emphasizes the importance of the position of the first relevant result in a list of retrieved documents, making it especially relevant for security patch tracing where we typically seek a single commit. MRR is defined mathematically as follows:

$$MRR = \frac{1}{|D|}\sum_{i=1}^{|D|}\frac{1}{\text{rank}_i} \tag{11}$$

In this equation, $|D|$ represents the total number of CVEs, and rank_i denotes the position of the first security patch for the i-th CVE. The MRR values range between 0 and 1, with higher values indicating better retrieval performance. By considering the inverse of the rank of the first relevant result, MRR encourages systems to prioritize the most relevant information at the top of the list, thus improving user satisfaction and system efficiency. The higher the MRR value, the better the security patch tracing performance.

Table 1: The effectiveness of PatchFinder and baselines to trace patch commits.

Recall@K	PatchScout	VCMatch	BM25	ColBERT	PatchFinder
K=1	46.25%	55.93%	11.88%	26.29%	**79.23%**
K=2	48.51%	57.72%	16.88%	31.49%	**79.30%**
K=3	48.72%	58.07%	19.58%	34.41%	**79.57%**
K=4	48.72%	58.42%	20.83%	37.23%	**79.64%**
K=5	48.92%	59.58%	22.08%	38.38%	**79.91%**
K=6	48.92%	61.88%	22.50%	40.93%	**79.97%**
K=7	48.92%	63.42%	23.54%	41.96%	**80.04%**
K=8	48.92%	63.42%	25.00%	43.22%	**80.31%**
K=9	48.92%	63.42%	25.83%	44.45%	**80.35%**
K=10	48.92%	63.42%	26.04%	44.95%	**80.63%**
MRR	0.3824	0.6195	0.1736	0.3240	**0.7951**

4.5.3 Manual Efforts@K. In the pursuit of tracing security patches within OSS, the Manual Efforts@K metric emerges as a classic metric. It represents the manual inspection effort required to locate the correct patch within the top-K results. If the desired security patch is found within these results, the effort corresponds to its rank. However, if the patch is not within the top-K, the effort is K, indicating a comprehensive search without success. Drawing from related work [39], the metric is mathematically expressed as:

$$\text{Manual Efforts@K} = \frac{\sum_{i=1}^{n}\min(\text{rank}_i, K)}{n} \tag{12}$$

A lower Manual Effort@K score is indicative of a more efficient and effective method for tracing security patch commits. This aids NVD security experts in mitigating the extensive manual work associated with tracing security patches, reducing inspection time, and enhancing patch detection accuracy.

5 Results and Discussion

We investigate the following research questions to provide a thorough analysis of the experimental results.

5.1 RQ1: Effectiveness Analysis

Table 1 and Table 2 show the effectiveness of different approaches in terms of Recall@K, MRR, and Manual Efforts@K, with the best one of each metric marked in bold. Table 1 reveals that PatchFinder significantly outperforms all the SOTA approaches across different values of K, for the Recall@K metric. The superiority of PatchFinder is most prominent at $K = 1$ where it achieves a Recall of 79.23%, markedly higher than PatchScout's 46.25%, VCMatch's 55.93%, BM25 's 11.88%, and ColBERT's 26.29%. This trend continues as K increases, showcasing the consistent effectiveness of PatchFinder in locating security patch commits within the top-K results. Notably, the Recall@K for PatchFinder remains above 79% for all values of K, highlighting its robustness in tracing relevant security patches. The MRR further confirms the effectiveness of PatchFinder with a score of 0.7951, significantly outpacing the 0.6195 and 0.3824 attained by VCMatch and PatchScout, respectively. This superior performance can be attributed to our two-phase approach that combines

Table 2: Manual Efforts@K (ME@K) of PatchFinder and baselines to trace security patch commits.

ME@K	PatchScout	VCMatch	BM25	ColBERT	PatchFinder	ME@K	PatchScout	VCMatch	BM25	ColBERT	PatchFinder
K=1	1.00	1.00	1.00	1.00	**1.00**	**K=8**	4.61	4.47	6.29	5.38	**2.38**
K=2	1.54	1.51	1.86	1.71	**1.20**	**K=9**	5.12	4.51	6.95	5.94	**2.58**
K=3	2.05	1.54	2.68	2.38	**1.40**	**K=10**	5.63	5.38	7.64	6.49	**2.77**
K=4	2.57	2.28	3.46	3.01	**1.60**	**K=20**	10.59	10.14	13.67	11.70	**4.71**
K=5	3.08	2.35	4.23	3.63	**1.80**	**K=30**	14.75	13.91	19.43	16.52	**6.65**
K=6	3.59	3.43	4.94	4.23	**1.99**	**K=50**	29.46	26.01	29.69	25.26	**10.52**
K=7	4.10	3.82	5.60	4.81	**2.19**	**K=100**	41.86	34.47	50.91	44.92	**20.21**

the lexical-level understanding from TF-IDF with the semantic understanding from the fine-tuned CR model. As shown in Table 2, PatchFinder requires much less manual effort compared to other methods. At $K = 1$, all methods tie with a score of 1.00, indicating minimal manual effort required regardless of the existence of true patches. However, as K increases, PatchFinder consistently requires less manual effort compared to the other baselines. For instance, at $K = 10$, PatchFinder registers a score of 2.77, which is considerably less than the manual effort demanded by the best baseline. This trend underscores the efficiency of PatchFinder, particularly as the value of K rises, demonstrating a lower manual effort requirement for practitioners aiming to trace security patches. Meanwhile, the observed lower effectiveness of PatchScout and VCMatch on our dataset compared to their published results [39, 43] can be attributed to two factors: ① *Data diversity:* As discussed before, PatchScout and VCMatch rely on handcrafted word-based similarities derived from their original dataset, which diminishes with increased data diversity. Such features struggle to capture relevant patch characteristics in a varied dataset. Our expanded dataset, featuring 4,789 CVEs from 532 OSS projects, presents an apparent contrast to their original dataset's 658 (1,669 for VCMatch) CVEs from only 5 (10 for VCMatch) OSS projects. This significant increase in both the number of CVEs and the diversity of originating projects introduces greater complexity, challenging their ability to accurately identify patches. ② *Language Specificity:* Their design primarily focuses on C/C++ OSS projects, which may not generalize well to other languages like Java and PHP, etc. These factors collectively lead to the variance in effectiveness between the original and our new datasets when applying PatchScout and VCMatch. Moreover, this comparison underscores an additional advantage of PatchFinder: unlike PatchScout and VCMatch, PatchFinder does not rely on predefined, language-specific features. This design enhances PatchFinder's scalability and applicability across various programming languages.

The observed results underscore the efficacy of PatchFinder in tracing security patch commits compared to existing baseline approaches. While previous ranking-based methods including PatchScout and VCMatch have made significant contributions when compared with match-based approaches [39, 43], their reliance on *handcrafted features* (predominantly *lexical only*). However, in practice, CVE descriptions and commits often use different terminologies to describe the same vulnerability including synonyms or varying phrasings. This discrepancy becomes particularly challenging when CVE descriptions or commits do not adhere to high-quality documentation standards, a situation frequently encountered with CVEs lacking associated patches. Consequently, their performance proves to be inadequate on the large and diverse projects that involve arbitrary descriptions and commits.

Table 3: Efficiency of PatchScout, VCMatch, and PatchFinder.

Tool	PatchScout	VCMatch	PatchFinder
Time Cost per CVE (s)	41.57	43.35	46.83

In contrast, PatchFinder leverages TF-IDF and CR to understand both the lexical and semantic aspects of the CVE description and commits, thereby achieving higher accuracy. Moreover, the fine-tuning process of CR allows our model to adapt to the specific semantics and patterns commonly found in security patches. This adaptability is a marked edge over methods such as PatchScout, which cannot adjust to the data on which they are deployed.

Efficiency Analysis. While PatchFinder employs a two-phase approach and involves fine-tuning CR, it maintains computational efficiency. To demonstrate this, we compared PatchFinder with PatchScout [39] and VCMatch [43] across our test set of 480 CVEs, conducting three trials to ensure accuracy. The results displayed in Table 3 reflect that PatchFinder's efficiency is closely competitive with the baselines. Despite a slightly higher overall time cost (3.48-5.26s), PatchFinder's performance is competitive, especially when considering its enhanced accuracy and scalability. Notably, Phase-2 required only 1.1 seconds on average per CVE. The initial retrieval phase consumes most of the time due to the extraction of lexical and semantic features. Specifically, the time cost for Phase-1 is 45.75 seconds per CVE. To speed it up, PatchFinder's efficiency can be easily enhanced by applying a vector database for retrieval, such as Faiss [26] or Redis [35] in this phase.

Answer to RQ1: *Compared with the four baselines, PatchFinder demonstrated superior effectiveness, achieving improvements of* 17.42%-54.59% *in Recall@10,* 0.1756-0.6215 *in MRR, and effectively reducing Manual Efforts@100 by* 14.26-30.7. *These results underscore the effectiveness of PatchFinder in patch tracing.*

5.2 RQ2: Ablation Analysis

In our proposed approach, we integrate two primary components: an initial retrieval using a hybrid retriever consisting of TF-IDF and pre-trained CodeReviewer (Phase-1), followed by a subsequent re-ranking using a fine-tuned CodeReviewer model (Phase-2). The input to our system encompasses the CVE description, commit messages, and code diffs. To dissect their effectiveness, we conducted a detailed ablation study by examining the performance of PatchFinder under various configurations: ① Using only the

Table 4: Contribution of individual components in PatchFinder in terms of Recall@K.

Recall@K	TD-IDF	$CR_{pretrain}$	Diff	Msg	Phase-1	Phase-2	PatchFinder
K=1	35.21%	28.75%	35.26%	63.85%	41.04%	0.21%	**79.23%**
K=2	40.21%	32.71%	42.60%	63.96%	47.71%	0.42%	**79.30%**
K=3	45.21%	34.58%	45.36%	64.06%	51.88%	0.42%	**79.57%**
K=4	47.71%	36.67%	46.67%	64.17%	54.38%	1.25%	**79.64%**
K=5	51.04%	38.33%	48.80%	64.27%	56.04%	1.46%	**79.91%**
K=6	52.71%	39.79%	49.27%	64.38%	57.50%	1.46%	**79.97%**
K=7	53.75%	40.21%	50.57%	64.48%	58.54%	1.46%	**80.04%**
K=8	54.79%	41.88%	52.08%	64.58%	59.17%	1.46%	**80.31%**
K=9	56.25%	43.13%	52.71%	64.58%	60.83%	1.46%	**80.35%**
K=10	57.29%	43.96%	55.70%	64.58%	61.46%	1.88%	**80.63%**
MRR	0.4243	0.3394	0.4146	0.6403	0.4827	-5.87E-06	**0.7951**

Table 5: Contribution of individual components in PatchFinder in terms of Manual Efforts@K (ME@K).

ME@K	TF-IDF	$CR_{pretrain}$	Diff	Msg	Phase-1	Phase-2	PatchFinder
K=1	1.00	1.00	1.00	1.00	1.00	1.00	**1.00**
K=2	1.54	1.71	1.51	1.25	1.59	2.00	**1.20**
K=3	2.05	2.38	1.54	1.51	2.11	2.99	**1.40**
K=4	2.57	3.04	2.28	1.76	2.59	3.99	**1.60**
K=5	3.08	3.67	2.35	2.02	3.05	4.98	**1.80**
K=6	3.59	4.29	3.43	2.27	3.49	5.96	**1.99**
K=7	4.10	4.89	3.82	2.53	3.91	6.95	**2.19**
K=8	4.74	5.49	4.61	2.78	4.33	7.93	**2.38**
K=9	5.19	6.07	5.12	3.03	4.73	8.92	**2.58**
K=10	5.63	6.64	5.63	3.29	5.13	9.90	**2.77**
K=20	9.52	11.94	10.59	5.83	8.63	19.67	**4.71**
K=30	12.76	16.57	14.75	8.37	11.66	29.20	**6.65**
K=50	18.58	24.73	29.46	13.45	16.96	47.81	**10.52**
K=100	31.19	43.02	41.86	26.16	27.90	93.07	**20.21**

lexical-based retriever, i.e., our TF-IDF retriever for tracing security patches (termed **"TF-IDF"**), ② Using only the semantic-based retriever, i.e., pre-trained CodeReviewer model for tracing security patches (termed **"$CR_{pretrain}$"**), ③ Utilizing only the CVE description and *commit messages* (termed **"Msg"**) for security patch tracing, and ④ Utilizing only CVE description and *code diffs* (termed **"Diff"**) for tracing, ⑤ Employing the initial retrieval phase only (termed **"Phase-1"**), and ⑥ Directly fine-tuning CodeReviewer without the first phase (termed **"Phase-2"**).

As shown in Table 4 and Table 5, the substantial improvement in terms of all metrics demonstrates the superiority of our two-phase approach over either a lexical-based or semantic-based retriever. PatchFinder attains a Recall@1 of 79.23%, which is more than double the performance when solely relying on TF-IDF or pretrained CodeReviewer. Notably, $CR_{pretrain}$, which has not undergone fine-tuning, still manages an acceptable score, especially when compared with ColBERT as shown in Table 1. This underscores the importance of domain-specific LLMs (CodeReviewer in PatchFinder) in understanding the underlying semantics in this task.

Similarly, both commit messages and code diffs play pivotal roles in capturing the nuanced semantics of commits, enabling accurate security patch tracing. The comparison suggests that commit messages have a good positive impact. This is likely because both CVE descriptions and commit messages are written in natural language, whereas code diffs are in various programming languages. This makes their lexical structures much less aligned, and thus relying solely on diffs in the initial phase can result in decreased recall. Nonetheless, commit messages and code diffs serve complementary roles in patch tracing within PatchFinder, rather than being mutually exclusive. Table 4 and 5 indicate that incorporating code diffs alongside commit messages significantly boosts the PatchFinder's effectiveness, demonstrating a notable improvement in Recall@1 from 63.85% to over 79.23% at Recall@1, an enhancement of 15.38%. It contributes to tracing an additional 16.05% of security patches that are untraceable by commit messages alone, and enhances the MRR by 0.1548, thereby reducing the manual effort by 5.95 commits for $K = 100$. This underlines the importance of code diffs in enriching our semantic analysis for more accurate patch identification.

Interestingly, based on our results in Table 1 and 2, TF-IDF tends to outperform BM25 for retrieving the commits that cover true patches. As explored in [18], BM25 employs a term saturation model and document length normalization for long texts. In our context, these aspects might inadvertently prioritize exact matches over partial ones, making it less effective for the concise nature of CVE descriptions and commits. Hence, weighing both effectiveness and efficiency, TF-IDF emerges as our preferred choice for this task.

Moreover, the quantitative results presented in Tables 4 and 5 clarify our motivation for adopting a two-phase design and demonstrate its essential role in addressing the challenges previously mentioned. Specifically, the **Phase-1** configuration effectively narrows down the dataset, ensuring the LLM in the re-ranking phase to focus its analysis on a more refined set of candidates. This is evident from its Recall@K, which reaches up to 61.46% at K=10, and an MRR of 0.4827, underscoring its critical contribution to ensuring broad coverage. Conversely, when the analysis is conducted with only **Phase-2**, its performance significantly drops with Recall@K peaking at merely 1.88% at K=10. This obvious under-performance highlights the LLM's challenges in dealing with vast, imbalanced datasets (with a patch to non-patch ratio of 1:5000), thereby illustrating the necessity of the initial retrieval phase in refining the dataset for subsequent LLM re-ranking. This refinement shifts the ratio from 1:5000 to a more manageable 1:100, ensuring focused and effective analysis.

Answer to RQ2: *Both main components and input of PatchFinder effectively contribute to the overall performance. Commit messages are important, while code diffs further complement and enhance semantic understanding. Meanwhile, the two-phase design of PatchFinder proves to be crucial for the overall effectiveness in locating patches, where only using one of the two phases can result in a significant decline in performance.*

5.3 RQ3: Distribution Analysis

This RQ focuses on a detailed investigation of the outcomes from PatchFinder. The primary aim is to explore if there exists a correlation between the types of vulnerabilities and the true patches identified by PatchFinder. Specifically, we are interested in examining whether PatchFinder's retrieval accuracy varies across different vulnerability categories and their respective severity levels.

To conduct this analysis, we categorized the vulnerabilities in our test dataset based on their Common Weakness Enumeration (CWE) identifiers [5] and Common Vulnerability Scoring System (CVSS) V2 scores [10]. Our analysis reveals that PatchFinder is particularly effective in tracing security patches for specific types of

vulnerabilities. It shows exceptional performance for vulnerabilities categorized under *CWE-125: Out-of-bounds Read* and *CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer*, achieving a tracing success rate of 100% (46/46) and 78.85% (41/52), respectively. While PatchFinder generally performs well across various CWE types, it does exhibit lower effectiveness for *CWE-122: Heap-based Buffer Overflow* and *CWE-834: Excessive Iteration*, with tracing ratios of 20% (1/5) and 33.3% (1/3), respectively. This indicates that the vulnerability distribution does have a significant impact on the effectiveness. The lower effectiveness observed for certain CWE types, such as CWE-122 and CWE-834, might be attributed to the inherent complexity of these vulnerabilities. For instance, heap-based buffer overflows (CWE-122) can manifest in various ways in the code, making them harder to trace even though using CR. Addressing this challenge might require more specialized features tailored to specific vulnerability types. Additionally, gathering more training data related to these challenging CWEs could enhance the model's understanding and improve performance.

We further investigated the relationship between the severity of vulnerabilities, as indicated by their CVSS V2 scores, and the level of difficulty in tracing their corresponding security patches. Our analysis reveals a notable correlation: vulnerabilities with higher CVSS scores are generally easier to trace. Specifically, PatchFinder successfully traced 84.52% (71/84) of high-severity CVEs. Surprisingly, the tracing success rate for medium-severity CVEs was slightly higher, at 86.09% (192/223). However, the tracing success rate for low-severity CVEs was the lowest, at 78.03% (135/173). One plausible explanation for this trend could be that high and medium-severity vulnerabilities often come with detailed descriptions, immediate developer attention, and increased community scrutiny, all of which facilitate more accurate tracing. In contrast, low-severity CVEs often receive less detailed documentation and lower levels of developer and community focus. Besides, the patches written by security analysts can differ according to the severity of vulnerabilities, which need dedicated strategies to capture their semantics. These findings highlight that the vulnerability severity involving the quality and characteristics of CVE artifacts is a non-negligible factor to consider for further refinements.

Answer to RQ3: *The vulnerability distribution does have a significant impact on the effectiveness. The performance is also tied to the severity of vulnerabilities, showing better outcomes for high and medium-severity vulnerabilities. Implementing specialized strategies and utilizing data augmentation could improve outcomes for more difficult types and vulnerabilities of low severity.*

5.4 RQ4: Practicality Analysis

In evaluating the practical utility of PatchFinder, we initially curated a dataset of 212, 074 CVE entries from NVD as of April 2023. However, we encountered a significant challenge in accurately identifying CVEs without patches since the "patch" tags in the NVD are often imprecise [39]: some entries with patches lack the corresponding tag, while others may be inaccurately tagged as having a patch but in fact, they do not.[1] Hence, we opted to focus on a more reliable subset: CVEs affecting OSS and known to lack patches. To this end, we leveraged the OSS project list maintained by OSS-Fuzz [13], resulting in 1,199 OSS projects. Further refining our selection, we excluded CVEs with any commit links on NVD, indicating the potential presence of a patch. This meticulous process yielded a targeted set of 473 CVEs, belonging to 268 OSS projects.

Upon deploying PatchFinder on this curated set, we derived the top-10 ranked outputs for each CVE. From this pool, we initially manually reviewed and traced 533 patches. These patches achieved a ranking of 1.65 in PatchFinder 's output on average. The entire review process was efficiently conducted, taking a total of 13.31 man-hours. We then submitted these patches to CNAs for review. Notably, 482 of these were subsequently confirmed by CNAs so far [46]. This achievement underscores the tangible benefits of PatchFinder and its potential to uncover and address omissions in current vulnerability databases.

Answer to RQ4: *PatchFinder effectively traces missing security patches for a significant set of CVEs in the NVD. From a curated set of 473 CVEs, PatchFinder's top-10 ranked outputs led to the identification of numerous plausible patch commits. Of the 533 manually reviewed patches (averaging a rank of 1.65), 482 were confirmed by CNAs, underscoring PatchFinder's practicality and its ability to address gaps in current vulnerability databases.*

5.5 Case Study

Among the 533 patches we traced in RQ4, a particularly illustrative instance is CVE-2022-31814 from the "pfSense-pfBlockerNG" project. As shown in Listing 2, this vulnerability allows remote attackers to execute arbitrary OS commands via specific manipulations. The associated patch commit, with its seemingly innocuous commit message "Update index.php" does not directly indicate its relevance to the CVE. This ambiguity poses challenges for tools like PatchScout and VCMatch who failed to recognize it. Their reliance on direct textual correlations (such as "# shared file names", "# shared function names", and "# shared words", etc.) between CVE descriptions and commits can be limited, especially when descriptions and commits employ synonyms or different phrasings.

In contrast, PatchFinder outperforms in such situations since harnessing the strengths of TF-IDF and CodeReviewer. Unlike other tools, it effectively deals with commits having limited information. Specifically, the patch commit in Listing 2 initially ranked 47th in our lexical-based retriever due to its brief message. However, the semantic-based retriever recognized its relevance, where the addition of `escapeshellarg` at Lines 23-24 crucially sanitizes shell metacharacters in the HTTP Host header, directly addressing the vulnerability. Such intricate changes, often missed by other tools, are accurately identified by PatchFinder due to its outstanding semantic analysis of code and text. As a result, our hybrid retriever improved its rank to 23rd. In the subsequent phase, the top-100 results, including this commit, were analyzed further. Here, the fine-tuned CR, adept at understanding code semantics, elevated its rank to 7th, placing it within the top-10 results.

5.6 Discussion on False Negatives

We further analyzed the missed patch commits of 95 CVEs and summarized them into three main causes.

- **Low-Quality CVE Descriptions (28/95):** Some CVE descriptions lack sufficient detail for effective patch tracing. Notably,

[1] There are 58.28% CVEs (123, 587/212, 074) missing "patch" tags in our initial dataset.

```
CVE Description:
pfSense pfBlockerNG through 2.1.4_26 allows remote attackers to
      execute arbitrary OS commands as root via shell
      metacharacters in the HTTP Host header. NOTE: 3.x is
      unaffected.
************************************************************
commit 071bdcf2d918c3e51cde11cf81fbd9b6f0379d7e
Author: BBcan177 <bbcan177@gmail.com>
Date:   Sun Jun 5 13:25:24 2022 -0400

    Update index.php

diff --git a/net/pfSense-pkg-pfBlockerNG/files/usr/local/www/
      pfblockerng/www/index.php
           b/net/pfSense-pkg-pfBlockerNG/files/usr/local/www/
      pfblockerng/www/index.php
index 8b8af0fab6b8..63f898b89246 100644
--- a/net/pfSense-pkg-pfBlockerNG/files/usr/local/www/pfblockerng
---/www/index.php
+++ b/net/pfSense-pkg-pfBlockerNG/files/usr/local/www/pfblockerng
+++/www/index.php
@@ -48,7 +48,7 @@ if (!empty($log)) {

 // Query DNSBL Alias for Domain List.
 $query = str_replace('.', '\.', htmlspecialchars($_SERVER['
      HTTP_HOST']));
-exec("/usr/bin/grep -l ' \"{$query} 60 IN A' /var/db/pfblockerng
-/dnsblalias/*", $match);
+exec("/usr/bin/grep -l " . escapeshellarg("\"{$query} 60 IN A")
+     . " /var/db/pfblockerng/dnsblalias/*", $match);
 $pfb_query = strstr($match[0], 'DNSBL', FALSE);
// Query for a TLD Block
```

Listing 2: The patch commit for CVE-2022-31814.

CVE-2022-0080 [29] only mentions "mruby is vulnerable to Heap-based Buffer Overflow", missing any vulnerability information except for vulnerability type.

- **Giant Commits (41/95):** ① *Irrelevant File Changes:* Commits such as [40] (a patch commit for CVE-2021-37686) often include changes unrelated to the patch, such as refactoring, which may hinder the patch's intent by introducing the noise. ② *Token Limit Exceedance:* Some patch commits (e.g., [24] for CVE-2017-18922) exceed CodeReviewer's token limit, affecting PatchFinder's ability to analyze them fully, even though we have pruned the code diffs before feeding them into CodeReviewer (as discussed in Section 3.2).
- **Confusing Commits (26/95):** Certain commits such as [17] deliberately obscure their patching role (fixing CVE-2017-13146 in this case). Still, non-patch commits within the same repository conversely claim it "fix" something, making it challenging for our hybrid retriever to identify candidate patches accurately.

6 Threats to Validity

External Threats. A primary external threat pertains to the reproducibility of the baselines. While we endeavored to faithfully implement PatchScout based on its published methodology, the absence of its source code posed challenges. To ensure a robust comparison with state-of-the-art methods, we also incorporated BM25 and ColBERT, which are notable sparse and dense retrieval models, respectively, as additional baselines.

Internal Threats. Our dataset's quality and scope could introduce internal threats. To minimize it, we initially built our dataset upon datasets from prior studies [39, 43], and followed their practice to source CVEs and security patches from various public advisories. Despite our efforts to curate a broad and diverse dataset, biases from these sources might persist. In the future, we will consider manual inspection or automatic techniques that can help assess and improve the data quality.

7 Related Work

There are numerous works focusing on tracing security patches, which can be divided into two categories: tracing security patches for disclosed vulnerabilities [27, 28, 39, 43, 48] and identifying silent security patches [1, 44, 47, 53–56].

Tracing security patches for disclosed vulnerabilities. Xu et al. [48] conducted an empirical study to understand the quality and characteristics of patches for disclosed vulnerabilities in two industrial vulnerability databases, thereby proposing to track patches from the CVE reference links across multiple knowledge sources (e.g., Debian). Their work focuses on analyzing reference links provided by security analysts, instead of directly tracing patches from OSS repositories. Tan et al. [39] conducted the most related work with PatchFinder. They designed a ranking-based tool named PatchScout to locate the patch commits by using RankNet on manually defined features from the CVE description and commits. Similarly, VCMatch [43], which directly classifies one commit as related or unrelated to the CVE description by fusing the features from PatchScout and extracted vectors from Bert. Unlike PatchScout and VCMatch, PatchFinder introduces a novel two-phase framework designed to overcome the challenges posed by large search spaces, and enables an end-to-end fine-tuning, to fully exploit the natural correlation between CVE descriptions and commits.

Identifying silent security patches. Besides, several works [1, 44, 47, 53–56] have delved into silent security patch identification. These efforts discern security patches but do not correlate them with specific vulnerabilities they rectify. In contrast, our focus is on tracing security patches tailored to a particular vulnerability, as defined by its CVE description.

8 Conclusion

In this paper, we present PatchFinder, an end-to-end and LLM-enhanced two-phase approach for effectively tracing security patches for disclosed vulnerabilities in OSS. The first phase employs a hybrid retriever for the initial retrieval of relevant commits, significantly narrowing down the candidate pool. The second phase leverages a fine-tuned CodeReviewer model to re-rank these commits, achieving a high degree of accuracy. Our extensive evaluations demonstrate that PatchFinder consistently outperforms state-of-the-art methods in Recall@K, MRR, and manual efforts, setting PatchFinder as a new benchmark in the field of security patch tracing.

Acknowledgments

This work is supported by the National Key R&D Program of China under grant 2021ZD0114501, the RIE2020 Industry Alignment Fund – Industry Collaboration Projects (IAF-ICP) Funding Initiative, as well as cash and in-kind contributions from the industry partner(s), the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] Rocío Cabrera Lozoya, Arnaud Baumann, Antonino Sabetta, and Michele Bezzi. 2021. Commit2vec: Learning distributed representations of code changes. *SN Computer Science* 2, 3 (2021), 150.

[2] Gobinda G Chowdhury. 2010. *Introduction to modern information retrieval*. Facet publishing.

[3] ClusterLabs/pacemaker. 2023. Fix: acl: Do not delay evaluation of added nodes in some situations · ClusterLabs/pacemaker@84ac07c · GitHub. https://github.com/ClusterLabs/pacemaker/commit/84ac07c. (Accessed on 10/09/2023).

[4] CVE. 2023. Home | CVE. https://www.cve.org/. (Accessed on 10/10/2023).

[5] CWE. 2023. CWE - CWE Glossary. https://cwe.mitre.org/documents/glossary/index.html. (Accessed on 10/12/2023).

[6] National Vulnerability Database. 2022. Spring4Shell: CVE-2022-22965. https://nvd.nist.gov/vuln/detail/cve-2022-22965. (Accessed on 31/01/2023).

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[8] Douglas Everson, Long Cheng, and Zhenkai Zhang. 2022. Log4shell Redefining the web attack surface. In *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2022*.

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv:2002.08155* (2020).

[10] Forum of Incident Response and Security Teams, Inc. 2023. Common Vulnerability Scoring System SIG. https://www.first.org/cvss/. (Accessed on 10/12/2023).

[11] GitHub. 2023. GitHub. https://github.com/. (Accessed on 10/13/2023).

[12] GitLab. 2023. GitLab. https://about.gitlab.com/. (Accessed on 10/13/2023).

[13] Google. 2023. google/oss-fuzz. https://github.com/google/oss-fuzz/tree/master/projects. (Accessed on 10/12/2023).

[14] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*. 933–944.

[15] Hao Guo, Sen Chen, Zhenchang Xing, Xiaohong Li, Yude Bai, and Jiamou Sun. 2022. Detecting and augmenting missing key aspects in vulnerability descriptions. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–27.

[16] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An empirical study on fine-tuning large language models of code for automated program repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1162–1174.

[17] ImageMagick. 2024. GitHub. https://github.com/ImageMagick/ImageMagick/commit/79e5dbcdd1fc2f714f9bae548bc55d5073f3ed20. (Accessed on 05/03/2024).

[18] Ammar Ismael Kadhim. 2019. Term weighting for feature extraction on Twitter: A comparison between BM25 and TF-IDF. In *2019 international conference on advanced science and engineering (ICOASE)*. IEEE, 124–128.

[19] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 6769–6781. https://doi.org/10.18653/v1/2020.emnlp-main.550

[20] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval* (Virtual Event, China) *(SIGIR '20)*. Association for Computing Machinery, New York, NY, USA, 39–48. https://doi.org/10.1145/3397271.3401075

[21] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 595–614.

[22] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]

[23] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating Code Review Activities by Large-Scale Pre-Training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1035–1047. https://doi.org/10.1145/3540250.3549081

[24] LibVNC. 2024. fix overflow and refactor websockets decode (Hybi) · LibVNC/libvncserver@aac95a9 · GitHub. https://github.com/LibVNC/libvncserver/commit/aac95a9dcf4bbba87b76c72706c3221a842ca433. (Accessed on 05/03/2024).

[25] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the vulnerability propagation and its evolution via dependency trees in the NPM ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*. 672–684.

[26] Meta. 2023. A library for efficient similarity search and clustering of dense vectors. https://github.com/facebookresearch/faiss. (Accessed on 10/12/2023).

[27] Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Xuan-Bach D Le, and David Lo. 2022. Vulcurator: a vulnerability-fixing commit detector. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1726–1730.

[28] Giang Nguyen-Truong, Hong Jin Kang, David Lo, Abhishek Sharma, Andrew E Santosa, Asankhaya Sharma, and Ming Yi Ang. 2022. Hermes: Using commit-issue linking to detect vulnerability-fixing commits. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 51–62.

[29] NVD. 2023. CVE-2022-0080. https://nvd.nist.gov/vuln/detail/CVE-2022-0080. (Accessed on 05/03/2024).

[30] NVD. 2023. NVD - CVE-2015-1867. https://nvd.nist.gov/vuln/detail/CVE-2015-1867. (Accessed on 10/09/2023).

[31] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 426–437.

[32] Yifan Qiao, Yingrui Yang, Haixin Lin, and Tao Yang. 2023. Optimizing Guided Traversal for Fast Learned Sparse Retrieval. In *Proceedings of the ACM Web Conference 2023*. 3375–3385.

[33] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[34] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[35] Redis. 2024. Redis. https://redis.io/. (Accessed on 05/03/2024).

[36] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[37] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[38] Snyk. 2023. Snyk Vulnerability Database. https://security.snyk.io/. (Accessed on 10/12/2023).

[39] 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 *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 3282–3299. https://doi.org/10.1145/3460120.3484593

[40] Tensorflow. 2024. Prevent a division by 0 in average ops. tensorflow/tensorflow@dfa22b3 · GitHub. https://github.com/tensorflow/tensorflow/commit/dfa22b348b70bb89d6d6ec0ff53973bacb4f4695. (Accessed on 05/03/2024).

[41] Torvald. 2023. torvalds/linux: Linux kernel source tree. https://github.com/torvalds/linux. (Accessed on 10/10/2023).

[42] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[43] Shichao Wang, Yun Zhang, Lingfeng Bao, Xin Xia, and Minghui Wu. 2022. VCMatch: A Ranking-based Approach for Automatic Security Patches Localization for OSS Vulnerabilities. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 589–600.

[44] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. 2021. Patchrnn: A deep learning-based system for security patch identification. In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*. IEEE, 595–600.

[45] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[46] Website. 2023. Website of Our Paper. https://sites.google.com/view/issta2024-patchfinder/home. (Accessed on 16/12/2023).

[47] Bozhi Wu, Shangqing Liu, Ruitao Feng, Xiaofei Xie, Jingkai Siow, and Shang-Wei Lin. 2022. Enhancing security patch identification by capturing structures in commits. *IEEE Transactions on Dependable and Secure Computing* (2022).

[48] 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* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 860–871. https://doi.org/10.1145/3540250.3549125

[49] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1385–1397.

[50] Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, Lingling Fan, Lida Zhao, Yiran Zhang, and Yang Liu. 2023. Mitigating persistence of open-source vulnerabilities in Maven ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 191–203.

[51] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675* (2019).

[52] Lida Zhao, Sen Chen, Zhengzi Xu, Chengwei Liu, Lyuye Zhang, Jiahui Wu, Jun Sun, and Yang Liu. 2023. Software composition analysis for vulnerability detection: An empirical study on Java projects. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 960–972.

[53] Jiayuan Zhou, Michael Pacheco, Jinfu Chen, Xing Hu, Xin Xia, David Lo, and Ahmed E. Hassan. 2023. CoLeFunDa: Explainable Silent Vulnerability Fix Identification. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2565–2577. https://doi.org/10.1109/ICSE48619.2023.00214

[54] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. 2021. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 705–716.

[55] Yaqin Zhou and Asankhaya Sharma. 2017. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 914–919.

[56] Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. 2021. Spi: Automated identification of security patches via commits. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–27.

Received 2024-04-12; accepted 2024-07-03