Efficient and Robust Security-Patch Localization for Disclosed OSS Vulnerabilities with Fine-Tuned LLMs in an Industrial Setting

Dezhi Ran
Key Lab of HCST (PKU), MOE; SCS, Peking University
Beijing, China
dezhiran@pku.edu.cn

Lin Li
Huawei Cloud Computing Technologies Co., Ltd.
Beijing, China
lilin88@huawei.com

Liuchuan Zhu
Huawei Cloud Computing Technologies Co., Ltd.
Beijing, China
zhuliuchuan1@huawei.com

Yuan Cao
School of EECS, Peking University
Beijing, China
cao_yuan21@stu.pku.edu.cn

Landelong Zhao
Huawei Cloud Computing Technologies Co., Ltd.
Beijing, China
zldl@stu.pku.edu.cn

Xin Tan
Huawei Cloud Computing Technologies Co., Ltd.
Beijing, China
tanxin50@huawei.com

Guangtai Liang
Huawei Cloud Computing Technologies Co., Ltd.
Beijing, China
liangguangtai@huawei.com

Qianxiang Wang
Huawei Technologies Co., Ltd
Beijing, China
wangqianxiang@huawei.com

Tao Xie*
Key Lab of HCST (PKU), MOE; SCS, Peking University
Beijing, China
taoxie@pku.edu.cn

Abstract

Security-patch localization, which links disclosed vulnerabilities in open-source software (OSS) to corresponding patches, has become a practical technique to mitigate the risk of OSS vulnerabilities in a timely manner. While existing approaches extensively focus on estimating the correlation between individual patches and Common Vulnerabilities and Exposures (CVEs), they often fail to address two major industrial requirements that make a tool of security-patch localization desirable in industrial settings: (1) efficiency when inspecting an enormous number of commits per vulnerability and (2) robustness to handle *confusing patches* (related but non-fixing commits). Toward addressing the preceding industrial requirements, in this paper, we report our experiences of developing and deploying Taper, a two-stage approach for efficiently and robustly locating security patches via mining the temporal relations among commits and CVEs. In the first stage, Taper extracts the information of the fixed version and the affected version from CVE descriptions to narrow down the inspection scope of commits, thus significantly improving the efficiency. In the second stage, Taper collects temporally co-located patches around the genuine security-patch commit as hard negative examples for security-patch localization. By fine-tuning a language model with these hard negative samples, Taper avoids recognizing confusing patches as security patches, thus improving patch-localization precision and robustness. We evaluate Taper against 2,128 CVEs from 978 OSS projects, which have a balanced distribution of programming languages and are consistent with industrial settings. Evaluation results show that Taper substantially outperforms a state-of-the-art approach named PatchFinder, improving the absolute MRR and Recall@1 by 0.422 and 0.541, respectively. Taper has been deployed at Huawei Cloud since October 2024. During 800 hours of operation, Taper helps locate over 52,140 security patches, providing daily service of security-patch localization for the Huawei company and Huawei Cloud users. We summarize three major lessons learned from developing and deploying Taper.

CCS Concepts

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

Keywords

Software Supply Chain, Patch Localization, Vulnerability Detection, Large Language Models, Software Security, Cybersecurity

ACM Reference Format:
Dezhi Ran, Lin Li, Liuchuan Zhu, Yuan Cao, Landelong Zhao, Xin Tan, Guangtai Liang, Qianxiang Wang, and Tao Xie. 2025. Efficient and Robust Security-Patch Localization for Disclosed OSS Vulnerabilities with Fine-Tuned LLMs in an Industrial Setting. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3696630.3728551

1 Introduction

With the wide adoption of Open-Source Software (OSS) in software industry, OSS vulnerabilities have also experienced rapid

*Tao Xie is the corresponding author.

FSE Companion '25, June 23–28, 2025, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728551

growth [2, 33] and resulted in severe losses [14, 33] due to their widespread use for building important applications [6, 50, 64]. While vulnerabilities can be fixed with *security patches* [10], over 90% of the vulnerabilities are fixed silently without a direct link to the corresponding security patches [52], making it difficult for developers to schedule software updates and missing valuable opportunities to understand security patches for subsequent reuse. To timely track silent fixes, *security-patch localization* [52] links disclosed OSS vulnerabilities, e.g., Common Vulnerabilities and Exposures (CVEs) [56], to the corresponding security patches.

To reduce the labor costs of security-patch localization, recent work [12, 31, 38, 52, 58] aims to automatically locate security patches by estimating the correlation between each patch and the CVE description based on lexical and semantic similarities. PatchScout [52] ranks software commits based on the correlation between CVE descriptions and commit messages using manually defined features with auxiliary information from CVE or National Vulnerability Database (NVD). VCMatch [58] improves upon PatchScout by learning semantic features in vulnerability descriptions and commit messages and using a voting-based rank fusion technique to combine results from multiple learned models for better performance. PatchFinder [31] incorporates code semantics through supervised fine-tuning over pairs of CVE descriptions and security patches to learn semantic similarity between commits and CVE descriptions.

While extensively focused on estimating the correlation between individual patches and CVEs, existing work [31, 52, 58] fails to meet two industrial requirements that make a tool of security-patch localization desirable in industrial settings.

Efficiency requirement for scanning commits at scale. Popular OSS projects typically contain an enormous number of commits, making security-patch localization akin to finding a needle in a haystack. Existing approaches [52, 58] employ a strategy of broad temporal search, examining all commits within an extended window around the CVE publication date. For instance, PatchScout [52] inspects commits within a two-year window (1.5 years before and 0.5 years after CVE release), while VCMatch [58] examines commits within a one-year window (0.5 years before and after). As shown in Figure 1, these exhaustive inspection strategies require examining over 10,000 commits to find the security patch, not only undermining effectiveness to find the correct security patch but also incurring substantial compute overhead.

Robustness requirement for distinguishing confusing patches from genuine security patches. Existing approaches [31, 52, 58] fall short in differentiating security patches from *confusing patches*, which share security-relevant characteristics with the actual fix but do not patch the given CVE. As illustrated in Figure 1, in addition to committing the security patch for CVE-2024-32002 (a path traversal vulnerability) in Git, developers also commit patches that add additional defense strategies against path traversal attacks. When locating security patches in this example, the state-of-the-art approach PatchFinder [31] fails to differentiate such confusing patches since it relies on a simplistic training strategy that contrasts known security patches against randomly selected non-security patches.

Toward satisfying the preceding industrial requirements for practical security-patch localization, in this paper, we report our experience of developing and deploying Taper, a two-stage approach designed based on two key insights. First, Taper substantially narrows down the inspection scope by leveraging version information in CVE descriptions. Version releases serve as critical milestones in OSS development, and CVE descriptions typically specify both the latest affected version and the version where the vulnerability is patched [12]. By extracting this version information and mapping it to corresponding tags in the OSS project under consideration, Taper substantially reduces the number of commits requiring inspection, helping address the efficiency challenge. Second, Taper exploits the observation that confusing patches usually co-occur with security patches, reflecting developers' concentrated efforts to defend against the revealed vulnerability. By utilizing these temporally co-located confusing patches as hard negative examples, Taper enhances its robustness and ability to distinguish between genuine security patches and related but non-fixing changes.

To instantiate the preceding insights into an automated, cost-effective, and robust system for industrial deployment, we implement Taper with fine-tuned Qwen-2.5 7B model [55] using two sources of data.

Robust version extraction with missing data handling. To address the challenge of inconsistent version-naming conventions in security advisories [11], we construct a dataset of 500 CVE descriptions paired with manually annotated version information in two steps. First, we extract the numeric version identifiers directly from CVE descriptions. Second, we transform these numeric versions to align with project-specific formatting conventions (e.g., adding "v-" prefixes where appropriate), making the output of an LLM directly usable for matching the affected software packages and versions. A key feature in our dataset is the explicit handling of incomplete information. For CVEs lacking version details in their descriptions, we deliberately label them as "None" rather than attempting to infer missing data. This approach significantly improves model robustness by teaching the LLM to recognize information gaps rather than hallucinating version numbers [24]. To ensure comprehensive coverage in production environments, we implement a fallback mechanism that activates when our model indicates that no version information is available. In these cases, Taper automatically employs established extraction techniques [52, 58]. For efficient model training, we employ Low-Rank Adaptation (LoRA) [21], a parameter-efficient fine-tuning technique updating only a small subset of model parameters. LoRA dramatically reduces computational requirements while maintaining extraction performance, making industrial deployment both practical and cost-effective.

Automated confusing-patch mining from temporal commit windows. Rather than relying on labor-intensive manual identification of confusing patches, we employ an automated data collection approach based on our observation that commits surrounding a security patch often address peripheral aspects of a vulnerability without providing the actual fix. These temporally co-located commits, exhibiting textual and structural similarities to genuine security patches, create significant challenges for robust security-patch localization. We collect the ten commits preceding and the ten commits following a confirmed security patch as negative examples in our training dataset. We fine-tune the LLM to distinguish between true security fixes and related but non-fixing modifications,

substantially improving the robustness of Taper in locating genuine security patches while avoiding false positives from confusing patches.

To evaluate the efficacy of Taper, we conduct extensive evaluations against a state-of-the-art approach PatchFinder [31] on security-patch localization datasets annotated from real-world OSS projects and CVE databases. Our results demonstrate Taper's effectiveness in three key aspects. First, Taper substantially outperforms PatchFinder, with 0.422 and 0.541 absolute improvement in Mean Reciprocal Rank (MRR) and Recall@1, respectively. Second, our version-based filtering reduces unnecessary patch inspections by over 90%, meeting industrial-deployment efficiency requirements. Third, our ablation studies confirm the critical importance of differentiating confusing patches. When using randomly selected commits as negative examples, Taper's performance actually decreases compared to fine-tuning with only positive security patch examples. In contrast, fine-tuning with temporally co-located commits as hard negative examples not only improves Taper's MRR and Recall@1 in security patch localization but also reduces manual inspection efforts by 68.1% when reviewing candidate security patches.

We have deployed Taper since October 2024 internally at Huawei Cloud [22], one of leading companies in cloud computing with over 3 million enterprise users and developers. Taper has run for over 800 hours and located security patches for more than 52,140 CVEs, which are incorporated into Huawei Cloud's vulnerability database as valuable security assets to support various vulnerability management services across the platform, directly enhancing the security posture of enterprise customers.

In summary, this paper makes the following main contributions:

- Taper, a novel two-stage approach to satisfy the efficiency and robustness requirements for security-patch localization in an industrial setting.
- Extensive evaluations on security-patch localization datasets annotated from real-world OSS projects and CVE databases, demonstrating the effectiveness, efficiency, and robustness of Taper.
- Experiences and lessons learned from developing and deploying Taper at Huawei Cloud, aligning future research on security-patch localization with realistic industrial settings.

2 Motivation

To streamline vulnerability management and ensure the security of provided services, software engineers in Huawei Cloud [22] aim to develop an approach to automatically identify security patches for a given CVE within a given OSS project. When localizing the security patch for fixing the vulnerability CVE-2024-32002 (depicted in Figure 1), the state-of-the-art approach PatchFinder [31] is faced with both **efficiency** and **robustness** challenges.

Efficiency Challenge. Figure 1 shows that the window between vulnerability discovery and disclosure spans thousands of commits. Existing approaches [52, 58] require scanning 1,000+ commits per CVE, prohibitively slow for Huawei Cloud's scale (thousands of repositories to analyze daily).

Robustness Challenge. As shown in Figure 1, three semantically similar but functionally distinct patches coexist: (1) the true security patch (middle), (2) a partial fix validating submodule directories while overlooking the symbolic link vulnerability (left), and (3) a post-fix enhancement adding clone-time hook validation (right). Current approaches [52, 58] fail to differentiate these cases because their training datasets lack such hard negatives as their training simply contrasts known security patches against random non-security patches, missing the subtle differences between security patches and confusing patches.

3 Taper Approach

3.1 Insights for developing Taper

To address the industrial challenges discussed in Section 2, we derive two key insights based on our industrial experiences.

Insight 1: Version information as natural boundary for efficiency improvement. CVE descriptions typically contain version information that delineates affected and patched versions of the CVE. This natural boundary can effectively narrow down the search space for candidate patches, significantly improving identification efficiency.

Insight 2: Leveraging co-occurring patches for robustness improvement against confusing patches. Security-relevant commits often cluster temporally, with actual security patches accompanied by related defensive changes. These co-occurring patches, while potentially confusing for automated tools, provide valuable training data for improving model robustness without additional labeling effort.

To instantiate the preceding insights, we design and implement a two-stage approach in Taper for security-patch localization. In the first stage, following the first insight, we develop a version-based filtering mechanism to narrow down candidate patches. In the second stage, guided by the second insight, we incorporate co-occurring patches into our training data to enhance model robustness against confusing patches.

3.2 Workflow of Taper

Figure 2 presents the workflow of Taper.

Inputs. The inputs of Taper consist of a reported CVE $\mathcal{CVE}$, the commit history $\mathcal{C} = \{c_0, ..., c_n\}$, and the list of tags in the affected OSS project.

Version-based candidate filtering. In the first stage, given the description of $\mathcal{CVE}$ and the list of tags, Taper prompts a fine-tuned LLM (detailed in Section 3.3) with the prompt described in Figure 3 to obtain the latest affected version and the first fixed version for filtering candidates to locate. Then Taper collects all the commits submitted between the two versions as C_{cand}, the candidate set for the second stage. If the LLM does not respond with the versions, Taper treats the version information as missing and falls back to the time-interval-based technique following previous work [52, 58], i.e., selecting all commits submitted within one year before the CVE is publicly published as C_{cand}, the candidate set for the second stage.

Generative localization of security patches. Given the candidate patches C_{cand} to inspect and the description of $\mathcal{CVE}$, Taper uses a fine-tuned LLM (detailed in Section 3.4) with the prompt shown in Figure 4 to judge whether each commit in C_{cand} is a security patch for the $\mathcal{CVE}$. Taper collects all commits for which

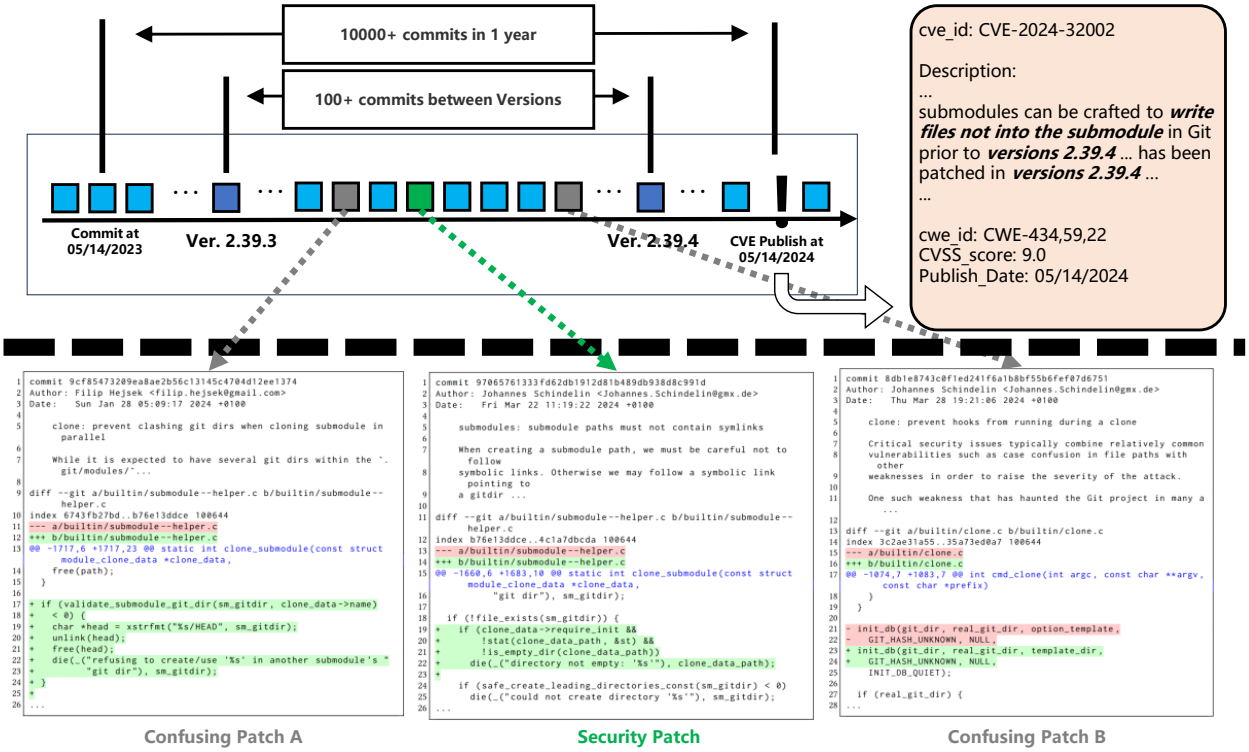

Figure 1: Locating the security patch for CVE-2024-32002, a path traversal vulnerability in Git's submodule handling. Top: Timeline showing that an vulnerability (allowing unauthorized file writes to ".git/" directory) was fixed in version 2.39.4 and publicly disclosed on 05/14/2024. Bottom: The actual security patch (green, middle) and two confusing patches that implement related defenses: one validating directory paths (left) and the other adding post-fix security enhancements (right).

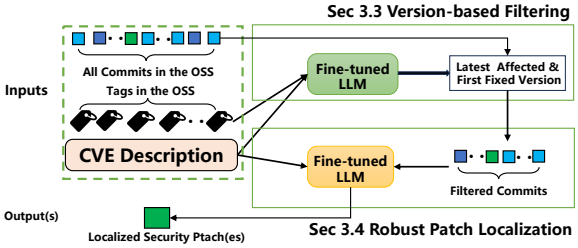

Figure 2: Workflow of using TAPER to locate security patches for a given CVE and an OSS project.

the LLM's responses are "Yes" as the located security patches for users to inspect.

Outputs. The outputs of TAPER are a set of identified commits $C_{cve} \subset C$ for manual examination. As shown in our evaluation (detailed in Section 4.4), the average size of C_{cve} is 1.73, substantially smaller than baseline approaches and making TAPER labor-economic for industrial localization of security patches.

3.3 Version-based Patch Filtering with A Fine-tuned Large Language Model

A popular OSS repository typically contains tens of thousands of commits, making it time-consuming to inspect all commits for each security-patch localization. While existing approaches [52, 58] filter commits based on CVE release timing (e.g., examining commits within 0.5 years before and after CVE publication), they still require inspecting over 1,000 commits to locate 75% of security patches (detailed in Section 4). Based on our experience at Huawei Cloud, over 90% of vulnerability descriptions contain version information (e.g., "prior to x.x.x" or "x.x.x was affected"), which can significantly narrow the search space. However, version information appears in diverse, non-standardized formats across vulnerability descriptions [11], making automated extraction of version information challenging.

To address these challenges, we exploit the power of large language models (LLMs) to cost-effectively extract version numbers from CVE descriptions and align them to the given OSS project. We construct a dataset of 500 CVEs published after 2022 with manually annotated version information through a two-stage process. First,

we extract numeric version identifiers directly from CVE descriptions. Second, we transform these identifiers to align with project-specific formatting conventions (e.g., adding "v-" prefixes where appropriate). Of the 500 CVEs, we successfully label version ranges for 462 (92.4%). For the remaining 38 CVEs lacking complete version information, we deliberately label them as "None" rather than attempting to infer missing data. This approach teaches the LLM to recognize information gaps rather than hallucinating version numbers [24]. The entire labeling process requires approximately 12 person-hours.

Using this dataset, we fine-tune a Qwen-2.5-7B model [55] using Low-Rank Adaptation (LoRA) [21], a parameter-efficient technique that updates only a small subset of model parameters while maintaining extraction performance. As shown in Figure 3, we create an instruction-tuning prompt [53] containing task instructions for version extraction, the CVE ID and the description, and the 300 most recent repository tags (automatically crawled from GitHub) to align version format consistency. This approach helps the LLM extract version information and align the version with project-specific version-naming conventions.

During inference, we use the same prompt template for extracting version information from unseen CVEs. To ensure comprehensive coverage in production environments, we implement a fallback mechanism that activates when our model indicates that no version information is available. In these cases, Taper automatically reverts to time-based filtering strategies following previous work [52, 58].

3.4 Robustness Fine-tuning Against Confusing Patches

As shown in Figure 1, patches near vulnerability fixes often make related code changes but do not directly address the security issue. These confusing patches frequently mislead existing approaches, significantly reducing localization accuracy. Improving robustness against such confusing patches requires additional fine-tuning, which can be time-consuming and labor-intensive if manual annotation is needed.

To address the preceding challenge cost-effectively, we propose a semi-automated approach to collect confusing patches. Our key insight is that OSS development exhibits temporal and spatial continuity, with related commits occurring consecutively. For each CVE, we collect N commits preceding and following the actual security patch as confusing patches, leveraging the GitHub API. These $2 \times N$ neighboring commits serve as negative samples for fine-tuning, where N is a configurable hyperparameter. Using the dataset of positive patches (actual security fixes) and negative samples (confusing patches), we fine-tune Qwen-2.5 7B using LoRA with the same settings described in Section 3.3. To enrich the context for LLM judgment, we augment each commit with its associated issue information, as issues often provide valuable complementary information about the commit's purpose. The final prompt template (Figure 4) combines CVE ID and its description, project name, commit details (hash, message, diff), and related issue information when available. When the assembled prompt exceeds the LLM's context length limit, we truncate the commit details to ensure that it fits within the allowed context window.

4 Evaluation

To evaluate the effectiveness of Taper, we conduct an extensive evaluation to investigate the following four research questions:

- **RQ1: Overall Effectiveness.** How effective is Taper compared with the state-of-the-art approach for security-patch localization?
- **RQ2: Efficiency Analysis.** How efficient is Taper in reducing the number of irrelevant patches?
- **RQ3: Robustness Analysis.** How robust is Taper against confusing patches?
- **RQ4: Ablation Study.** How does an individual component contribute to the effectiveness of Taper?

4.1 Evaluation Setup

4.1.1 Datasets. To evaluate Taper for real-world security-patch localization, we curate a dataset from two sources: CVEfixes [5], and our industrial deployment at Huawei Cloud. We have obtained 1,825 CVEs and their fixes from the CVEfixes dataset while we have also manually collected fixes for 303 CVEs ourselves. All vulnerabilities are identified post-2018 to avoid the pitfalls of earlier vulnerabilities that often have informal and less valuable descriptions. To enhance real-world applicability, we minimize reliance on CVE Reference URLs (for patch links), which are crucial for identifying patches for vulnerabilities that lack existing links. Moreover, by employing regular expressions, we extract issues from commit messages containing #NUM, correlating them with corresponding issue information within the repository to obtain issue titles and descriptions, which are then included in prompts. In all, our dataset consists of 2,128 CVEs and their fixes. We randomly take 1,928 of them for fine-tuning our model and take the remaining 200 CVEs as the test set. For constructing hard negative samples, the training set selects commits adjacent to positive samples, specifically 10 before and after. For the testing set, we include all commits between tags as outlined in the vulnerability description. If version information is absent, we randomly select 150 commits between the vulnerability's disclosure date and the positive sample's submission date, using the GitHub Compare API to retrieve all relevant commits within specified tag intervals.

4.1.2 Implementation of Taper. We implement Taper in Python. We choose two open-source LLMs Llama3.1 [54] and Qwen2.5 [60], as the base pre-trained LLMs, which possess balanced performance on instruction understanding and code-related tasks. During fine-tuning, the batch size is set to 1 and the maximum epoch is 4. We adopt Adam [28] as the optimizer with a learning rate of 5e-5. All preceding hyper-parameters are determined based on the validation set by selecting the best ones among some alternatives. All experiments are conducted on a server with 64 CPU cores (Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz), 512 GB RAM, and 8 Huawei Ascend 910B GPUs (64 GB memory each).

4.1.3 Baseline. We compare Taper with the state-of-the-art security-patch localization approach PatchFinder [31]. PatchFinder first employs the TF-IDF [8] and pre-trained CodeReviewer [32] to calculate similarity score among candidate candidates and filter out the lower ones in its initial retrieval. It then fine-tunes an LLM, specifically CodeReviewer, to rank the filtered commits through relevance

```
### Task
Read the vulnerability description carefully, select the most appropriate
tag numbers from the tag list as the earliest fixed version and latest
affected version of the vulnerability.
### Input
Vulnerability {CVE_ID}: {CVE_Description}
Which has been confirmed affects: {Repo_Path}
Here is the list of tags in the above code repository: {Tag_List}
### Notes
Note that the list may not be complete, but it is guaranteed to be
continuous, so select from the list whenever possible.
### Response Format
Your return should follow a JSON format with two keys: 'first_fixed_tag'
and 'last_affected_tag'.
---
Label: first_fixed_tag:{First Fixed Tag Value},
last_affected_tag:{Last Afftecte Tag Value}
```

Figure 3: Prompts Used for Version Extraction.

```
### Input
Vulnerability {cve_id}: {description}
Which has been confirmed affects: {repo_path}
Here's the detail of a commit of the repository.
```json
{detail}
```
And the issue related to this commit
```json
{issue_info}
```
Task
Could this commit be the fix patch of {cve_id}?'.

Label: {Yes Or No}
```

Figure 4: Prompts Used for Security-patch Localization.

Table 1: Overall Effectiveness

Approach	MRR	Recall@1	Recall@2	Recall@3	Recall@4	Recall@5
PatchFinder [31]	0.149	0.0%	8.5%	11.5%	14.0%	16.0%
PatchFinder$_{Fine-tuned}$ *	0.229	0.0%	13.5%	20.5%	26.0%	29.0%
Taper	0.651	54.5%	67.5%	71.5%	73.3%	74.0%

* We fine-tune the model of PatchFinder with the same training set as Taper.

scores. In our experiments, we strictly follow PatchFinder's code and guide to train it on our training and testing datasets, which are exactly the same datasets where we train and test Taper in the experiments.

4.1.4 Evaluation Metrics. Although Taper is not a ranking-based approach for security-patch localization, we measure Taper with the same metric as a ranking-based approach [31] to fairly compare with the baseline and evaluate Taper's effectiveness. Taper predicts whether a candidate patch is the security patch or not, and ranks all patches with prediction 'Yes' ahead of 'No'.

Recall@K is a widely used metric in retrieval and ranking systems. It is calculated as the ratio of the positive samples found in the top-K results to the total number of positive samples among the candidates. In rare cases, Taper predicts multiple candidate patches to be the security patch and the metric is calculated as expectation values when randomly ranking these positive patches.

Mean Reciprocal Rank (MRR) focuses more on the highest-rank positive samples, ranging from 0 to 1, with higher value standing for better localization ability. It calulates the sum of the reciprocal of their rank as follows:

$$MRR = \frac{1}{|D|} \sum_{cve \in D} \frac{1}{\max_{c \in C_{cve}} \text{rank}_c} \tag{1}$$

, where D represents the evaluation dataset, $rank_c$ is patch c's ranking in results, and cve and C_{cve} stand for a CVE description and its corresponding positive samples, respectively. Note that we have only one positive sample $c_{cve} \in C_{cve}$, so it can be written as follows:

$$MRR = \frac{1}{|D|} \sum_{cve \in D} \frac{1}{\text{rank}_{c_{cve}}} \tag{2}$$

Similar to calculating Recall@K, when Taper predicts multiple candidate patches to be the security patch, the metric is calculated as expectation values.

4.2 RQ1: Overall Effectiveness

Table 1 presents the effectiveness comparison between Taper and PatchFinder, revealing three key findings.

First, Taper achieves substantially higher MRR (0.651) compared to both the original PatchFinder (0.149) and its fine-tuned variant (0.229), and demonstrates consistently superior performance across all Recall@K metrics. Second, Taper notably achieves 54.5% Recall@1 compared to PatchFinder's 0%. This dramatic enhancement in early-stage recall shows that Taper can precisely tell which candidate is the actual security patch, while PatchFinder fails to distinguish the confusing patches. Third, even after fine-tuning PatchFinder with our training set, the substantial performance gap persists, highlighting the fundamental advantages of our approach. Compared to baseline approaches, Taper's design assures its robustness against confusing patches.

We investigate the significant performance gap between PatchFinder and Taper, as well as PatchFinder's performance drop compared to its reported results. Our analysis reveals two fundamental factors that explain PatchFinder's limited effectiveness even after fine-tuning. First, our evaluation dataset better represents contemporary

Table 2: Programming Language Distribution in Our Dataset

Language	C/C++	GO	Java	PHP	Python	JavaScript	TypeScript	Others
Training set	418	289	263	264	241	164	104	185
Testing set	62	5	43	24	20	21	7	18
Total	480	294	306	288	261	185	111	203

security-patch localization scenarios in real industry environment. As shown in Table 2, our dataset achieves more balanced coverage across programming languages, whereas PatchFinder's original dataset is heavily skewed toward C/C++ repositories (88%). Moreover, our dataset exhibits significantly better repository diversity. PatchFinder's original dataset has 41% of samples concentrated in just five repositories (1% of total repositories), potentially limiting its generalizability. Second, PatchFinder's original training strategy fails to learn robust security-patch patterns. PatchFinder randomly selects 5,000 non-patch commits in the code repository as negative samples (for each CVE), which are not strong enough for model to learn how to distinguish confusing patches from the actual security patches. This training-data disparity makes the model less effective under real industrial demands. These limitations underscore the importance of Taper's design choices to distinguish confusing patches.

4.3 RQ2: Effectiveness of Patch Filtering

We compare the version-based filtering algorithm with the patch-filtering algorithms used in existing approaches PatchScout [52] and VCMatch [58]. PatchScout collects all commits in a project that was made from 1.5 years before to 0.5 years after the public disclosure of a vulnerability. This design is to capture a wide range of potentially relevant changes, including those that may not be immediately connected to the vulnerability but could influence the development context or subsequent fixes. VCMatch collects all commits (from the repository) that occurred within 0.5 years before and after the vulnerability's release. This tighter focus aims to hone in on commits more directly related to the vulnerability's introduction and initial mitigation efforts.

Substantial efficiency improvement over existing approaches. Figure 5 and Figure 6 present the statistics of the number of candidate commits filtered after Taper and time-interval-based filtering employed by baseline approaches, respectively. By leveraging version information, our approach demonstrates significant efficiency in filtering relevant commits compared to traditional time-based filtering approaches. The median number of candidate commits is 36 for Taper, which is substantially lower than VCMatch (481 commits) and PatchScout (880.5 commits).

Effectiveness of fine-tuning for version extraction. Table 3 shows Taper's fine-tuning models' substantial improvements over baselines without fine-tuning in terms of the accuracy of version extraction. The non-fine-tuned baselines, which lack the benefit of learned patterns from historical data, fall short in accurately identifying version-specific commits, even for developed commercial models such as GPT-4o-mini. However, both LLaMa3.1-8B and Qwen2.5-7B achieve very high accuracy after fine-tuning. Especially, LLaMa3.1 has a relatively weaker instruction-following ability before fine-tuning (0.035) but improves drastically after fine-tuning (0.882). It is worth noting that we deliberately keep about 10% of the CVEs where no exact version is given in the description in the dataset. In these cases, we find that our fine-tuned model can report this situation instead of generating hallucinations.

Table 3: Effectiveness of Fine-tuning for Version Extraction.

LLMs	Accuracy
GPT-4o-mini	0.684
Qwen2.5-7B-Instruct	0.455
Qwen2.5-7B-Instruct-Lora-SFTed	**0.893**
LLaMa3.1-8B-Instruct	0.035
LLaMa3.1-8B-Instruct-Lora-SFTed	**0.882**

4.4 RQ3: Robustness Against Confusing Patches

In this RQ, we investigate the contribution of contrastive fine-tuning over hard negatives (i.e., the confusing patches) mined from the neighbor of security patches.

Table 4 presents detailed ablation results for different confusing-patch construction approaches, revealing the following critical insights about the effectiveness of our design choices.

The baseline results without confusing patches demonstrate the necessity of confusing-patch construction in our approach. Using Qwen-2.5-7b without fine-tuning achieves only moderate performance with an MRR of 0.455 and Recall@1 of 32.3%. Our neighbor-based confusing-patch construction strategy shows remarkable improvements over both the baseline and random sampling approaches. With N=10 neighbors, our approach achieves an MRR of 0.651 and Recall@1 of 54.5%, representing substantial improvements of 250% and 183%, respectively, compared to random sampling with 40 samples (MRR=0.401, Recall@1=29.7%).

The impact of neighborhood size (N) reveals interesting patterns about our approach's robustness. Performance remains consistently strong across different neighborhood sizes, with N=10 showing slightly optimal results. When N=5, the model achieves nearly comparable performance (MRR=0.641, Recall@1=53.2%), with only marginal decreases of 1.5% in MRR and 2.4% in Recall@1 when compared to N=10. Increasing to N=20 also shows slight performance drop, along with a 10.4% increase in average commits identified (1.91 vs 1.73). This pattern suggests that the construction of confusing-patch samples, rather than the quantity of confusing-patch samples, is the key factor in improving model discrimination. The stability of performance across different neighborhood sizes also suggests that our approach is both robust and computationally efficient, requiring only a small number of carefully selected confusing-patch samples to achieve optimal results.

Noticeably, the stark contrast between neighbor-based and random sampling approaches is particularly evident in the progression of random sampling results. Even as we increase the number of random samples from 10 to 40, the performance gains are modest (MRR improves from 0.305 to 0.401, Recall@1 from 21.4% to 29.7%), while still maintaining unacceptably high false positive rates. This comparison validates our hypothesis that temporally proximate commits provide more valuable confusing-patch examples for model training than randomly selected samples, leading to both better discrimination and more efficient patch identification, suggesting

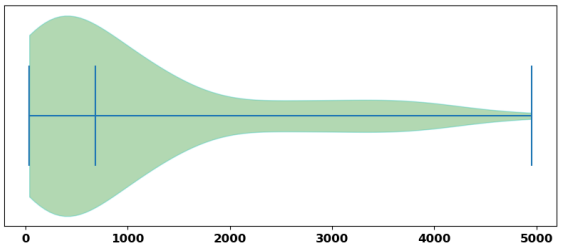

Figure 5: Distribution of the Number of Commits to be Inspected by Existing Approaches.

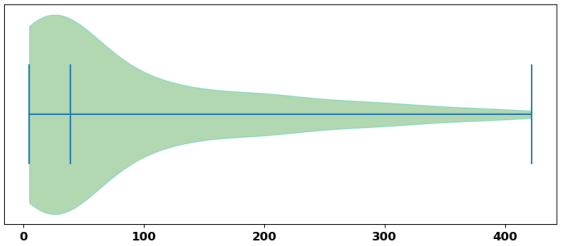

Figure 6: Distribution of the Number of Commits to be Inspected by Taper. Note that the X-axis scales are different.

Table 4: Ablation on Different Confusing Patch Settings with Qwen-2.5-7B model.

Confusing Patch Construction	MRR	Recall@1	Recall@2	Recall@3	Avg. commits *
No confusing patches	0.455	32.3%	52.1%	61.7%	5.41
Neighbors-with-N=5	0.641	53.2%	67.9%	71.6%	1.74
Neighbors-with-N=10	0.651	54.5%	67.5%	71.5%	1.73
Neighbors-with-N=20	0.629	51.8%	67.1%	71.5%	1.91
Ramdomly-pick-10-samples	0.305	21.4%	33.4%	40.4%	30.66
Ramdomly-pick-20-samples	0.345	24.4%	38.4%	45.6%	21.18
Ramdomly-pick-40-samples	0.401	29.7%	44.0%	51.5%	17.15

* "Avg. commits" represents the average number of commits identified by Taper as the security patches for each CVE, i.e., the average number of commits that need manual inspection.

that our approach is both robust and computationally efficient, requiring only a small number of carefully selected confusing-patch samples to achieve optimal results.

More concerning is the high number of false positives, with an average of many commits identified per CVE indicating the model's limited ability to discriminate security patches without proper confusing-patch samples. The effectiveness of our approach is further evidenced by the dramatic reduction in false positives. Our approach identifies an average of only 1.73 commits per CVE, showing a dramatic reduction compared to 5.41 commits with the baseline, 17.15 commits with random-40 sampling, 21.18 with random-20, and 30.66 with random-10 sampling. This dramatic reduction in false positives without sacrificing recall makes our approach valuable for practical scenarios of security-patch localization. In real industry, we need to check only less than one false positive commit manually per CVE on average, while other approaches require several dozens times of the labor force.

4.5 RQ4: Ablation Study on LLMs

In this RQ, we investigate the impact of using different LLMs for instantiating Taper and the impact of fine-tuning LLMs with domain-specific data, with results presented in Table 5.

Fine-tuning consistently and substantially outperforms prompt engineering for small models. Our results demonstrate that fine-tuning approaches consistently yield better results compared to prompt engineering across all model sizes and architectures. Taking the LLaMa3.1-8b model as an example, the fine-tuned version (LoRA-fine-tuning-with-N=10) achieves an MRR of 0.596 and Recall@1 of 48.6%, while the prompting-only version achieves lower scores with an MRR of 0.456 and Recall@1 of 33.9%. Similarly, for Qwen2.5-7b, fine-tuning improves MRR from 0.455 to 0.651 and Recall@1 from 32.3% to 54.5%, representing relative improvements of 40.9% and 64.7% respectively. This performance gap remains consistent across all evaluation metrics, with the fine-tuned model showing substantial improvements in higher-K recall values (e.g., Recall@5: 74.0% vs 69.8% for Qwen2.5-7b). While prompt engineering can help guide models toward better performance, the improvements are modest compared to the substantial gains achieved through fine-tuning. This finding is particularly evident in security-patch localization tasks where subtle code patterns and complex security implications need to be understood. Fine-tuning allows models to learn these nuanced patterns through repeated exposure to relevant examples, whereas prompt engineering alone struggles to effectively encode the domain knowledge. The consistent performance advantage of fine-tuned models emphasizes the importance of task-specific model adaptation through fine-tuning for specialized tasks such as security-patch localization.

Fine-tuning smaller models consistently and substantially outperforms larger models without fine-tuning. Our experimental results reveal a compelling finding: fine-tuned smaller models achieve significantly better performance than larger models relying solely on prompting. Specifically, our fine-tuned Qwen2.5-7b model achieves an MRR of 0.641 and Recall@1 of 53.2%, substantially outperforming GPT-4o mini (MRR=0.501, Recall@1=37.6%) despite the latter being a larger model. The performance advantage of the fine-tuned model extends across all recall thresholds, with particularly notable gains in early recall metrics (Recall@2: 67.9%

Table 5: Ablation Results on LLMs.

Model	Training	MRR	Recall@1	Recall@2	Recall@3	Recall@4	Recall@5
GPT-4o mini	Prompting	0.501	37.6%	56.7%	65.6%	70.2%	73.2%
LLaMa3.1-8b	Prompting	0.456	33.9%	49.9%	57.6%	61.9%	64.6%
Qwen2.5-7b	Prompting	0.455	32.3%	52.1%	61.7%	66.7%	69.8%
LLaMa3.1-8b	LoRA-fine-tuned	0.596	48.6%	63.3%	66.2%	67.7%	68.8%
Qwen2.5-7b	LoRA-fine-tuned	0.651	54.5%	67.5%	71.5%	73.3%	74.0%

vs 56.7%, Recall@3: 71.6% vs 65.6%). This finding demonstrates that domain-specific fine-tuning can effectively compensate for model-size limitations. The superior performance can be attributed to the fine-tuning process allowing smaller models to develop specialized knowledge about security patch characteristics, while larger models, despite their broader knowledge base, lack this targeted expertise. This finding has important practical implications for industrial deployment, as it suggests that organizations can achieve better results using smaller, fine-tuned models, which are more computationally efficient and cost-effective than using larger foundation models' API service out of the box.

5 Deployment and Lessons Learned

5.1 Deployment of Taper

We have deployed Taper at Huawei Cloud since October 2024 to locate security patches for disclosed OSS vulnerabilities continuously. The system runs on two main parts: a processing system and an LLM inference service. We use Huawei Cloud's Container Engine, a customized Kubernetes cluster service with 120 Docker containers to process OSS commits in parallel. The LLM inference service runs on four Huawei Ascend 910B GPU cards, each with 64GB of Memory. Taper has run for over 800 hours and processed more than 52,140 CVE vulnerabilities. The security team manually checks all patches that Taper finds and adds them to Huawei Cloud's vulnerability database. The verified patches are now key security assets that support various vulnerability management services [23]. One key use case is Huawei Cloud's third-party vulnerability-mitigation service, which tells cloud users whether their systems have been affected by known security issues and suggests specific library updates to fix these problems.

5.2 Lessons Learned

5.2.1 Costs Matter in Industrial Deployment. Beyond algorithm innovations, our deployment experience reveals that both runtime and monetary costs significantly impact the practical adoption of security-patch localization in industrial settings.

Runtime costs matter. Our evaluation (Section 4.3) demonstrates that traditional approaches [12, 31, 52] become computationally prohibitive at scale. These approaches, which compute similarities between CVE descriptions and repository commits, face significant challenges given the exponential growth in both vulnerability reports (over 40,000 CVE numbers in 2024 alone [47]) and repository commits (Figure 5). To reduce runtime costs, we introduce version-based filtering, a simple yet effective technique that reduces the median number of commits requiring analysis from hundreds to fewer than 50 per vulnerability, as shown in Figure 6.

Trade-offs between runtime costs and infrastructure support. Our experience shows that infrastructure optimization can effectively address efficiency requirements. Instead of relying on rate-limited GitHub APIs [16], we maintain local mirrors of open-source repositories. While this approach requires substantial storage resources, the caching strategy enables Taper to process hundreds of vulnerabilities continuously in production, demonstrating how infrastructure investments can break runtime cost barriers.

Monetary costs matter. The cost considerations extend beyond computational resources to model selection and deployment. Proprietary LLMs, while powerful, incur significant operational costs in industrial deployment. Our solution involves fine-tuning and deploying small-scale open-source LLMs locally, providing a more cost-effective alternative while maintaining acceptable performance. This approach demonstrates how careful consideration of monetary constraints can guide technical decisions without compromising system effectiveness.

5.2.2 There is a Need to Embrace Open-source LLMs and Data Engineering for Software Development Our experience with Taper highlights two key advantages of LLMs over traditional deep-learning approaches for industrial software engineering: (1) reduced dependency on machine-learning expertise, and (2) alignment with open-source ecosystems. Simultaneously, it underscores a paradigm shift toward data-centric engineering in LLM-driven development.

LLMs enable typical software developers to bypass machine learning expertise. Traditional deep-learning approaches [3] impose significant barriers, requiring developers to master both complex machine learning concepts for model-architecture selection [48] and hyper-parameter tuning [62]. In contrast, LLMs provide a more accessible alternative that aligns with existing software development practices. This accessibility proves particularly valuable in industrial settings where software teams may lack specialized machine-learning expertise but possess strong software-engineering capabilities.

A thriving open-source community simplifies the development and deployment of LLMs. Our deployment experience demonstrates that developers can effectively utilize LLMs by focusing primarily on prompt engineering and dataset preparation, rather than managing complex deep-learning model configurations. The thriving open-source ecosystem, featuring powerful models and mature deployment systems such as vLLM [29] and SGLang [65], significantly reduces implementation barriers. These tools enable

teams to achieve production-quality performance through fine-tuning without extensive deep-learning expertise, supporting industrial priorities of rapid development and maintainable systems.

General-purpose models need adaptation for downstream tasks. While LLMs offer significant advantages, they are not plug-and-play solutions for specific software-engineering tasks [40, 41]. Our experiences (Table 4 and Table 5) demonstrate that successful implementation still requires attention to data preparation [66] and prompt engineering [27]. This finding suggests a paradigm shift: while LLMs reduce the complexity of model development, they increase the importance of data and prompt quality management [42]. This shift aligns well with traditional software engineering practices, where data-pipeline development and quality control are familiar concepts.

5.2.3 There is a Need to Align Research Benchmarks with Industrial Settings. To develop practical tools for software engineering, it is important to ensure the alignment between research benchmarks and real-world industrial settings. Our industrial deployment revealed a significant gap between research evaluation practices and real-world settings in two major aspects. First, as demonstrated in Section 2, confusion patches, which frequently occur in industrial settings, are largely absent from current research datasets [12, 31, 52]. For example, existing approaches [31, 52] randomly selects 5,000 commits as negatives and one security commit as a positive, circumventing the confusing patches and the key challenge. Our experience shows that these confusion patches can significantly impact tool effectiveness in practice, and yet their impact remains under-evaluated in academic research. Second, we observe that the common practice of single-branch patch labeling in research datasets poorly reflects industrial reality. Many software projects maintain multiple stable branches, each receiving security patches with different commit IDs [51]. While existing research typically considers only patches from a single branch as ground truth, our deployment experience shows that valid patches often exist across different branches. This mismatch between evaluation methodology and practical requirements can lead to misleading conclusions about tool effectiveness. These findings underscore the importance of developing more representative evaluation datasets better capturing the complexity of industrial environments.

6 Related Work

6.1 Security-patch Localization for Disclosed Vulnerabilities

Localization of security patches for disclosed vulnerabilities has been extensively studied with various techniques applied, including SVM [39], boosting [12], RankNet [52], Transformers [38], and recently LLMs [31, 63]. PatchScout [52] and VCMatch [58] extract features from CVE and code commits, ranking the most related code commit among 5000 randomly selected commits. PatchFinder [31] first applies an initial retrieval to decrease the number of candidate pairs of commits and CVEs by comparing cosine similarity of their TF-IDF [8] vectors and their encoding in CodeReviewer [32], and then utilizes a linear classifier on each pair's encodings to predict how likely they are corresponding. However, all the preceding approaches assume that the negative examples are randomly sampled without considering confusing patches, being unrealistic for practical security-patch localization [36].

Complementing existing work, Taper explores how to fulfill the efficiency and robustness industrial requirements for security-patch localization, validates the importance of differentiating confusing patches, and aligns future research with real industrial settings.

6.2 Large Language Models for Software Engineering

Large language models (LLMs) has been widely adopted for software-engineering tasks [19, 26, 41, 42] such as code generation [7, 17, 61] and test generation [9, 40, 43, 46]. Existing work exploits LLMs in two primary approaches. First, prompt engineering [35] carefully designs prompts, i.e., the inputs to the LLMs, to tailor model responses for specific tasks [9, 59]. Recent studies have shown that prompt engineering can significantly improve the performance of LLMs on tasks such as code generation [13, 25, 30], and even more complex tasks such as code explanation [1, 15] and security issues [18]. Despite widely adopted due to its convenience, prompt engineering falls short in tasks requiring in-depth comprehension and knowledge beyond LLMs' inherent ability on code-related tasks [49]. Second, fine-tuning LLMs on domain-specific datasets has emerged as a potent approach to adapt general-purpose models to specialized software-engineering tasks [67]. CodeLLaMa [44], as a representative with state-of-the-art performance, enhance LLaMa's code generation ability with fine-tuning on multiple downstream tasks. To reduce the costs of fine-tuning, parameter-efficient fine-tuning (PEFT) approaches [20, 21, 34] have been particularly successful in adapting models for complicated downstream applications and outperform full fine-tuning (FFT) approaches in code-understanding tasks [4], e.g., automated code review [37] and code summarization [45, 57]. Taper adopts LoRA [21] for fine-tuning LLMs on security-patch localization, achieving satisfactory performance with low computational resources.

7 Conclusion

In this paper, we have reported our experiences and lessons learned from developing and deploying Taper for security-patch localization. Compared to state-of-the-art approaches, Taper exploits the version information in CVE descriptions to drastically reduce the number of commits to inspect, and fine-tunes a pre-trained language model using confusing patches that are temporally correlated with the security patches, meeting the efficiency and robustness requirements for industrial localization of security patches. The evaluation results demonstrate the effectiveness of Taper, improving the MRR of security-patch localization from 0.229 to 0.641 while avoiding inspection of over 90% unnecessary commits. We have deployed Taper at Huawei Cloud and summarized three lessons learned from developing and deploying Taper.

Acknowledgments

This work was partially supported by National Natural Science Foundation of China under Grant No. 623B2006 and Grant No. 92464301. We thank Professor Wei You and Dr. Hao Yu for their insightful comments.

References

[1] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. 2024. Automatic semantic augmentation of language model prompts (for code summarization). arXiv:2304.06815

[2] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2023. Empirical analysis of security vulnerabilities in Python packages. *Empirical Software Engineering* (2023).

[3] Md Zahangir Alom, Tarek M. Taha, Christopher Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Brian C Van Esesn, Abdul A S. Awwal, and Vijayan K. Asari. 2018. The history began from AlexNet: A comprehensive survey on deep learning approaches. arXiv:1803.01164

[4] Shamil Ayupov and Nadezhda Chirkova. 2022. Parameter-efficient finetuning of transformers for source code. arXiv:2212.05901

[5] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*.

[6] Simon Butler, Jonas Gamalielsson, Björn Lundell, Christoffer Brax, Anders Mattsson, Tomas Gustavsson, Jonas Feist, Bengt Kvarnström, and Erik Lönroth. 2022. Considerations and challenges for the adoption of open source components in software-intensive businesses. *Journal of Systems and Software* (2022).

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, and et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374

[8] Gobinda Chowdhury. 2010. *Introduction to Modern Information Retrieval.*

[9] 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.*

[10] Nesara Dissanayake, Asangi Jayatilaka, Mansooreh Zahedi, and M. Ali Babar. 2022. Software security patch management - a systematic literature review of challenges, approaches, tools and practices. *Information and Software Technology* (2022).

[11] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. 2019. Towards the detection of inconsistencies in public security vulnerability reports. In *Proceedings of the 28th USENIX Security Symposium.*

[12] Trevor Dunlap, Elizabeth Lin, William Enck, and Bradley Reaves. 2024. VFCFinder: Pairing security advisories and patches. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security.*

[13] Jean-Baptiste Döderlein, Mathieu Acher, Djamel Eddine Khelladi, and Benoit Combemale. 2023. Piloting Copilot and Codex: Hot temperature, cold prompts, or black magic? arXiv:2210.14699

[14] Douglas Everson, Long Cheng, and Zhenkai Zhang. 2022. Log4shell: Redefining the web attack surface. In *Proceedings of the 2022 Workshop Meas., Attacks, Defenses Web.*

[15] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2023. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. arXiv:2304.11384

[16] GitHub. 2025. GitHub REST API documentation. https://docs.github.com/en/rest?apiVersion=2022-11-28

[17] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, and et al. 2024. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence. arXiv:2401.14196

[18] Jingxuan He and Martin Vechev. 2023. Large language models for code: security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security.*

[19] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* (2024).

[20] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. arXiv:1902.00751

[21] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685

[22] Huawei. 2025. Corporation information of Huawei. https://www.huawei.com/en/corporate-information

[23] Huawei. 2025. Huawei CodeArts Inspector. https://www.huaweicloud.com/product/vss.html

[24] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, and et al. 2023. Survey of hallucination in natural language generation. *Comput. Surveys* (2023).

[25] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. SelfEvolve: A code evolution framework via large language models. arXiv:2306.02907

[26] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2025. From LLMs to LLM-based agents for software engineering: a survey of current, challenges and future. arXiv:2408.02479

[27] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. 2023. Challenges and applications of large language models. arXiv:2307.10169

[28] Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *In Proceedings of the 2014 International Conference on Learning Representations* (2014).

[29] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with PagedAttention. arXiv:2309.06180

[30] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2024. AceCoder : An effective prompting technique specialized in code generation. *ACM Transactions on Software Engineering and Methodology* (2024).

[31] 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.* 590–602.

[32] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, and et al. 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.*

[33] Ruyan Lin, Yulong Fu, Wei Yi, Jincheng Yang, Jin Cao, Zhiqiang Dong, Fei Xie, and Hui Li. 2024. Vulnerabilities and security patches detection in OSS: A survey. *Comput. Surveys* (2024).

[34] Haokun Liu, Derek Tam, Muqeeth Mohammed, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. In *Proceedings of the 2022 Neural Information Processing Systems.*

[35] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys Home* (2023).

[36] Xueqing Liu, Jiangrui Zheng, Guanqun Yang, Siyan Wen, and Qiushi Liu. 2025. Improving the context length and efficiency of code retrieval for tracing security vulnerability fixes. arXiv:2503.22935

[37] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. LLaMA-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning . In *Proceddings of the 2023 IEEE 34th International Symposium on Software Reliability Engineering.*

[38] 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.*

[39] 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 *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering.*

[40] Dezhi Ran, Hao Wang, Zihe Song, Mengzhou Wu, Yuan Cao, Ying Zhang, Wei Yang, and Tao Xie. 2024. Guardian: A runtime framework for LLM-based UI exploration. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis.*

[41] Dezhi Ran, Mengzhou Wu, Yuan Cao, Assaf Marron, David Harel, and Tao Xie. 2025. An infrastructure software perspective toward computation offloading between executable specifications and foundation models. *Science China Information Sciences* (2025).

[42] Dezhi Ran, Mengzhou Wu, Wei Yang, and Tao Xie. 2025. Foundation model engineering: Engineering foundation models just as engineering software. *ACM Trans. Softw. Eng. Methodol.* (2025).

[43] Dezhi Ran, Mengzhou Wu, Hao Yu, Yuetong Li, Jun Ren, Yuan Cao, and et al. 2025. Beyond pass or fail: Multi-dimensional benchmarking of foundation models for goal-based mobile UI navigation. arXiv:2501.02863

[44] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, and Sten Sootla et al. 2024. Code Llama: Open foundation models for code. arXiv:2308.12950

[45] Iman Saberi, Fatemeh Fard, and Fuxiang Chen. 2024. Utilization of pre-trained language models for adapter-based knowledge transfer in software engineering. *Empirical Software Engineering* (2024).

[46] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 1 (2023).

[47] SecurityScorecard. 2025. Number of CVEs per year. https://www.cvedetails.com/browse-by-date.php.

[48] Mohit Sewak, Sanjay Sahay, and Hemant Rathore. 2020. An overview of deep learning architecture of deep neural networks and autoencoders. *Journal of Computational and Theoretical Nanoscience* (2020).

[49] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. 2023. Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks. arXiv:2310.10508

[50] Diomidis Spinellis and Vaggelis Giannikas. 2012. Organizational adoption of open source software. *Journal of systems and software* (2012).
[51] Xin Tan, Yuan Zhang, Jiajun Cao, Kun Sun, Mi Zhang, and Min Yang. [n. d.]. Understanding the practice of security patch management across multiple branches in OSS projects. In *Proceedings of the ACM Web Conference 2022*.
[52] 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*.
[53] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.
[54] Meta-Llama Team. 2024. Introducing Llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/
[55] Qwen Team. 2024. Qwen2.5: A party of foundation models. https://qwenlm.github.io/blog/qwen2.5/
[56] Common Vulnerabilities. 2005. Common vulnerabilities and exposures. *The MITRE Corporation* (2005).
[57] Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. 2023. One adapter for all programming languages? Adapter tuning for code search and summarization. In *Proceedings of the 45th International Conference on Software Engineering*.
[58] Shichao Wang, Yun Zhang, Liagfeng Bao, Xin Xia, and Minghui Wu. 2022. VCMatch: A ranking-based approach for automatic security patches localization for OSS vulnerabilities. In *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering*.
[59] Mengzhou Wu, Hao Wang, Jun Ren, Yuan Cao, Yuetong Li, Alex Jiang, Dezhi Ran, Yitao Hu, Wei Yang, and Tao Xie. 2024. Skill-adpative imitation learning for UI test reuse. arXiv:2409.13311
[60] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, and et al. 2025. Qwen2.5 technical report. arXiv:2412.15115
[61] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. CoderEval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*.
[62] Tong Yu and Hong Zhu. 2020. Hyper-parameter optimization: A review of algorithms and applications. arXiv:2003.05689
[63] Junwei Zhang, Xing Hu, Lingfeng Bao, Xin Xia, and Shanping Li. 2024. Dual prompt-based few-shot learning for automated vulnerability patch localization. In *Proceedings of the 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering*.
[64] Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, and et al. 2023. Mitigating persistence of open-source vulnerabilities in Maven ecosystem. arXiv:2308.03419
[65] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient execution of structured language model programs. In *Proceedings of the 38th Annual Conference on Neural Information Processing Systems*.
[66] Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, and et al. 2023. LIMA: Less is more for alignment. In *Proceedings of the 37th Conference on Neural Information Processing Systems*.
[67] Wentao Zou, Zongwen Shen, Qi Li, Jidong Ge, Chuanyi Li, Xiang Chen, Xiaoyu Shen, LiGuo Huang, and Bin Luo. 2025. Experimental evaluation of parameter-efficient fine-tuning for software rngineering tasks. *ACM Transactions on Software Engineering and Methodology* (2025).