LLM-SZZ: Novel Vulnerability-Inducing Commit Identification Driven by Large Language Model and CVE Description

Siqi Fan[1], Xin Liu[1†], Yingli Zhang[1], Yuan Tan[1], Luxing Yin[1], Zhaorun Chen[3],
Song Li[2], Lei Qiao[1], Rui Zhou[1]
{fansq19, bird, 120220909270, tany18, tr0jan, fly2moon, zr}@lzu.edu.cn,
songl@zju.edu.cn, zhaorun@uchicago.edu
[1]School of Information Science and Engineering, Lanzhou University, Lanzhou, China
[2]The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China
[3]Department of Computer Science, University of Chicago, Chicago, United States

***Abstract*—The SZZ method and its variants are widely employed to identify vulnerability-affected ranges by analyzing vulnerability-fixing commits to trace back vulnerability-inducing commits. However, these methods generally suffer from low precision due to several key factors: 1) Current static method-based variants often incorrectly consider too many irrelevant lines and files in a commit. While methods that extract file references from vulnerability discussions can help narrow down relevant files, obtaining bug discussions for every CVE is often difficult. 2) Learning-based approaches focus exclusively on code to capture semantic relationships for identifying root cause lines. However, these models utilize limited information and demonstrate insufficient capacity for effective capture. 3) The reliance on line mapping algorithms results in inadequate tracing capabilities for complex vulnerabilities, especially when vulnerability-inducing commits are obscured in earlier software versions.**

To address these issues, this paper innovatively incorporates semantic information from descriptive text and the nature of CVEs derived from vulnerability-fixing commit diffs. By leveraging large language models (LLMs), this approach aims to capture the true root cause lines of vulnerabilities more accurately and enhance the tracing capabilities of the SZZ method, thereby achieving precise localization of the vulnerability impact range.

Experimental results indicate that our proposed LLM-SZZ method outperforms existing state-of-the-art approaches, achieving over a 18% increase in precision across datasets in various programming languages, demonstrating a significant performance advantage.

***Index Terms*—SZZ, Vulnerability, Common Vulnerabilities and Exposures, Large Language Model**

I. Introduction

The remediation for security vulnerabilities is not only time-consuming but also costly. An example is Log4Shell (CVE-2021-44228 [1]), a severe Remote Code Execution (RCE) vulnerability discovered in Apache Log4j. Despite Apache's prompt release of patches to address this vulnerability, the extensive incorporation of Log4J across global software supply chains prolongs the impact. According to the U.S. Department of Homeland Security, identifying and remedying all affected instances may take at least a decade [2].

While the Common Vulnerabilities and Exposures (CVE) system provides a unified identifier to facilitate the sharing of vulnerability information, the details included in CVE entries often lack accuracy and completeness. Based on the research by Bao et al., their analysis of 172 vulnerability samples revealed that the version ranges recorded in the National Vulnerability Database (NVD) differed from the actual ranges for 99 of those samples [3]. Such discrepancies can lead security teams to misallocate precious protective resources to system versions that are not actually at risk and may result in security experts overlooking certain vulnerabilities during the remediation process, leaving systems vulnerable to potential cyber-attacks.

To accurately identify the range of software versions affected by bugs, the SZZ method was initially proposed in the research [4]. Its core idea involves searching for vulnerability-fixing commits in the historical versions of software and using these commits to reverse-track and identify vulnerability-inducing commits, thus determining the origin of bugs. Specifically, the SZZ method leverages annotation or blame tools in version control systems to analyze the modified lines of source code in vulnerability-fixing commits, enabling the tracing of the original commits that may have caused the bugs.

To enhance the accuracy of the SZZ method, various improved variants have been introduced in subsequent research. These variants can be classified into two categories: The first focuses on filtering out noise in vulnerability-fixing commits to accurately identify root cause lines, which includes both static methods [5–8] and learning-based approaches [9]. For example, a recent study [8] has demonstrated that the inclusion of files referenced in bug discussions can enhance the effectiveness of SZZ in identifying vulnerability-inducing commits by addressing tangled commits [10]. However, this method merely extracts files mentioned in the bug discussion, rather

This work is supported by the National Key R&D Program of China under Grant No. 2024YFE0203800, the Open Research Fund of The State Key Laboratory of Blockchain and Data Security, Zhejiang University, and the Supercomputing Center of Lanzhou University.

†Corresponding Author

than incorporating the relevant semantic information related to the vulnerabilities. Moreover, only a small number of CVEs have detailed bug discussions, making this approach clearly unsuitable for widespread adoption. NEURAL-SZZ [9] utilizes a heterogeneous graph attention network (HAN) to identify the root cause line. While the authors successfully reduced the false positive rate, this achievement came at the expense of significantly decreasing the number of correctly identified vulnerability-inducing commits, indicating that solely extracting the semantic meanings of changed lines with HAN is insufficient.

The second category aims to improve the capability to trace the earliest commits that modified the vulnerable lines [3]. The V-SZZ [3] method introduced the concept of multiple backtraces, using line mapping algorithms to find vulnerabilities in early versions of software. However, its backtracking capability remains inadequate, and it cannot locate the root cause line in vulnerability-fixing commits, leading to a high false positive rate that renders it impractical for real-world use.

To address the noise issue discussed above, this paper proposes the LLM-SZZ method as a novel approach. Firstly, we employ a Large Language Model (LLM) to analyze natural language information from CVE descriptions and the associated vulnerability-fixing commit diffs to fully comprehend the nature of the vulnerability and identify the root cause deletion line. Next, LLM-SZZ integrates line mapping with LLM analysis to facilitate multiple backtraces, thereby identifying vulnerability-inducing commits present in earlier versions. Finally, the method implements a final selection process from the candidate vulnerability-inducing commits, effectively addressing the complexities associated with tangled commits.

The main contributions of this paper can be summarized as follows:

- We analyzed the shortcomings of existing methods and demonstrated that their noise filtering and backtracking strategies are insufficient for identifying inducing commits.
- We propose the first approach that leverages LLMs with customized prompting strategies for identifying vulnerability-inducing commits, achieving significant improvements in precision over prior methods.
- We designed a series of experiments demonstrating that the incorporation of semantic information from CVE descriptions and vulnerability-fixing commit diffs can aid in the tracing of inducing commits.
- We compared our approach with state-of-the-art methods using a real-world dataset that has previously been employed in SZZ methods, facilitating straightforward comparisons and demonstrating the effectiveness of LLM-SZZ.

II. Motivation

Utilizing static methods to identify root cause deletion lines is often insufficient, as these methods frequently overlook lines relevant to vulnerabilities while failing to exclude unrelated lines. For instance, CVE-2017-1000398 [11] is a vulnerability in Jenkins that permitted unauthorized users to access information about running tasks on agents via the remote API. As shown in Fig. 1, the associated vulnerability-fixing commit involves three modified files, with the actual vulnerability-inducing commit identified by tracing back the deletion line 485 in *Executor.java*, resulting in commit *bf3016*. We employed B-SZZ [4], MA-SZZ [6], and AG-SZZ [5] to identify the vulnerability-inducing commit for this CVE. Upon manual analysis, we found that both MA-SZZ and B-SZZ failed to filter out any irrelevant lines. AG-SZZ recognized the modifications in *Executor.java* and *WorkUnit.java* as non-semantic lines. However, it ultimately overlooked the true root cause line.

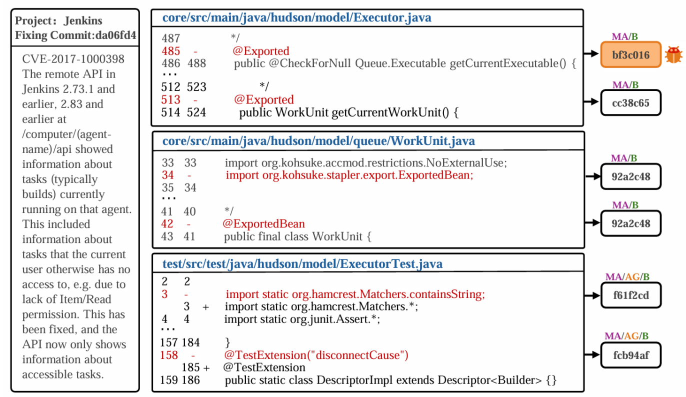


Fig. 1: Example of the Affected Range Identification Process Using Static Methods

This example also clearly illustrates that these methods struggle to address tangled commits (i.e., a single commit containing multiple unrelated files). FI-SZZ [8] claims to assist in locating vulnerability-related files by extracting files mentioned in bug discussions. However, it is evident that this CVE does not have a bug discussion report, which directly prevents its application in this case.

Most existing SZZ methods lack sufficient backtracking capability. Similar to the aforementioned approaches, they can only perform a single backtrace, whereas V-SZZ is the first to introduce multiple backtracking capabilities. Fig. 2 illustrates the processes of V-SZZ in handling CVE-2016-10049 [12], a buffer overflow vulnerability. The fixing commit utilizes the *magickmax()* function to filter out inputs that could lead to a buffer overflow.

In the case of deleted line A, the first backtrace leads to commit *2174484*, terminating further tracing due to the line mapping algorithm. While tracking deleted line B, V-SZZ performs five consecutive backtraces until it identifies commit *7131d8f* as a stopping point. Consequently, V-SZZ identifies the vulnerability-inducing commits as *2174484* and *7131d8f*. However, it still falls short in pinpointing the actual vulnerability-inducing commit, *3ed852*, highlighting the limitations of its line mapping algorithm in addressing complex or deeply embedded code vulnerabilities.

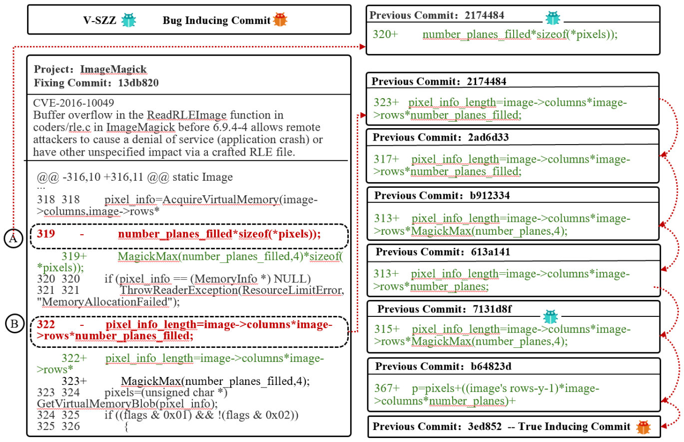


Fig. 2: Example of Affected Range Identification Process of V-SZZ

Additionally, V-SZZ conducts backtracing on all deletion lines within the vulnerability-fixing commits, designating all identified results as vulnerability-inducing commits. Previous evaluations stipulate that if any of the identified vulnerability-inducing commits is indeed the actual one, the identification is considered correct. This practice significantly overestimates the method's performance in real-world applications, leading to a sharp increase in the false positive rate.

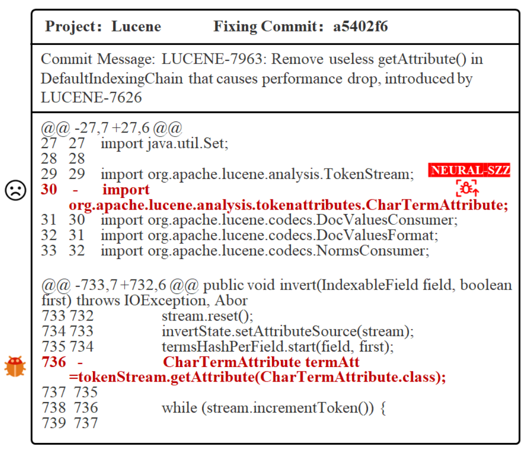


Fig. 3: Example of Root Cause Line Identification of NEURAL-SZZ

We also examined a case study involving NEURAL-SZZ. Fig. 3 presents a simple yet typical example from the Lucene project, where a useless call to *getAttribute()* in the *DefaultIndexingChain* causes a performance drop. In this example, it is necessary to select one root cause line from two deletion lines. However, NEURAL-SZZ erroneously identifies deletion line 30, which is clearly unreasonable. It fails to accurately capture the semantic meaning of deleted statements. Considering the advanced language comprehension capabilities of large models, we submitted this issue to ChatGPT, providing the commit message and the fixing commit diff as input. The response from ChatGPT is illustrated in Fig. 4.

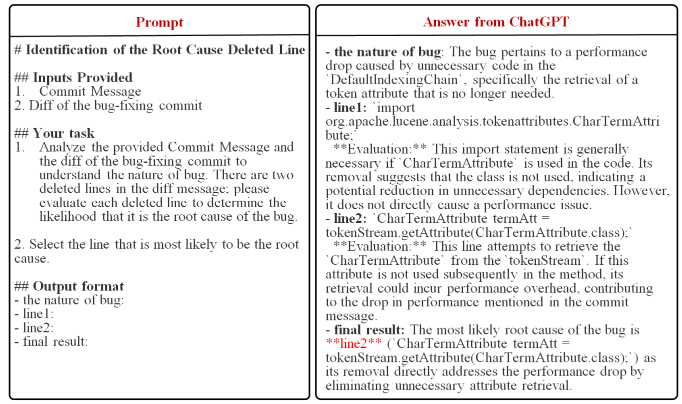


Fig. 4: Example of Root Cause Line Identification of ChatGPT

ChatGPT demonstrates a remarkable ability to comprehend vulnerabilities and accurately identify root cause deletion lines. This highlights the significant potential of LLMs in analyzing bug information and code. With its robust natural language processing capabilities and extensive domain knowledge in security and software, the LLM is well-equipped to achieve a profound understanding of vulnerabilities through limited natural language input. Additionally, the model excels at interpreting the code-to-comment alignment (associating code parts with their corresponding comments or descriptions), allowing it to establish connections between natural language and code effectively [13, 14].

We found that in this case, providing the LLM with both commit messages and fixing-commit diff information results in a clearer understanding of the vulnerabilities compared to supplying only the fixing-commit diff information.

Based on these observations, this paper presents LLM-SZZ, which explores the potential of LLMs to enhance the accuracy of root cause line identification and improve vulnerability tracking capabilities.

III. The Design of LLM-SZZ

Inspired by the case described in Section II, we propose a new method named LLM-SZZ, as illustrated in Fig. 5. This approach initially utilizes LLM to understand CVE descriptions and fixing commit diffs, aiming to combine more information to capture the semantic relationship. With these insights, LLM-SZZ identifies the root cause line within a vulnerability-fixing commit, then traces back through each previous commit and filters out irrelevant files from tangled commits, thereby enabling a deeper trace back into the origins of the vulnerability. To address the instability of LLM outputs, we also apply a voting strategy that selects the majority response from up to five runs.

Similar to the study of V-SZZ [3], we call the immediately previous commits to the lines changed in the fixing commit as the *previous commits* and all the commits that previously modified the lines changed in the fixing commit as the *descendant commits*.

The LLM-SZZ method is structured into three steps:

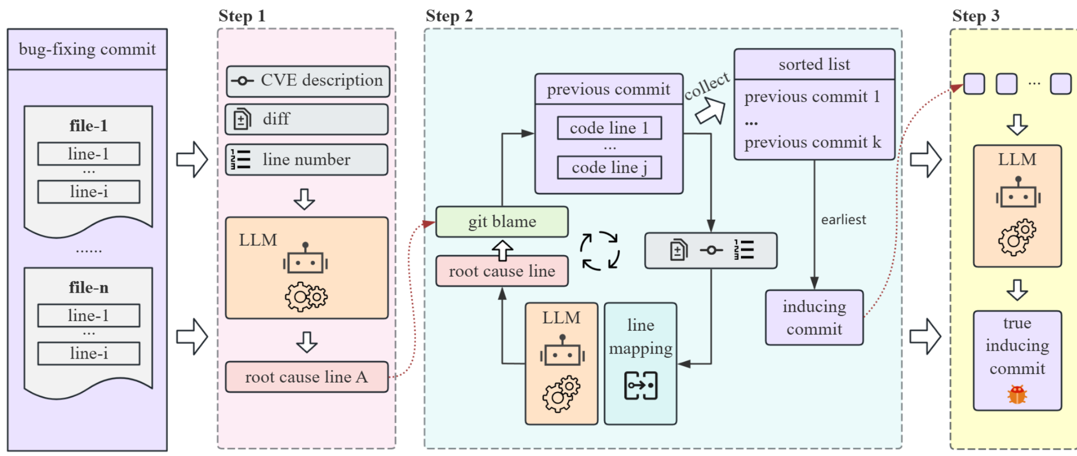


Fig. 5: The Workflow of LLM-SZZ

1) **Identification of the Root Cause Deleted Lines**: Utilize LLM to analyze CVE descriptions and vulnerability-fixing commit diffs. By aligning natural language information with code, this process facilitates a clearer understanding of the CVE and enables the identification of the root cause line for each file involved in the vulnerability-fixing commit.
2) **Vulnerability Backtrace Decision:** First, use *git blame* to identify the previous commit associated with the root cause lines. Once the previous commits are identified, a combination of a line mapping algorithm and analysis by LLM determines whether to trace descendant commits. If continuation is warranted, the root cause line in the previous commit is selected for further traceback. This iterative process persists until a decision to cease the traceback is reached.
3) **Optimal Candidate Selection:** Utilize LLMs that have been provided with information on CVE descriptions and vulnerability-fixing commit diffs to evaluate candidate-inducing commits associated with each file, selecting the most likely commit as the true vulnerability-inducing commit.

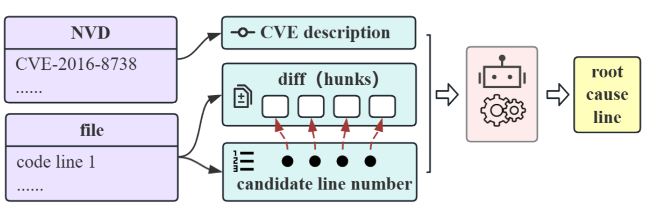


Fig. 6: Identification of the Root Cause Deleted Line

A. Identification of the Root Cause Deleted Line

SZZ methods presume that statements involved in a vulnerability are deleted while those fixing the vulnerability are added. To trace the origins of a vulnerability and study its evolutionary process, deleted lines serve as critical clues. Guided by this principle, our first step in the method is to identify root cause lines that have been removed in a commit. A vulnerability-fixing commit may involve multiple files, each containing several deleted lines. Simultaneously considering all deleted lines from all files could overwhelm the input capacity of LLM and potentially reduce accuracy. Therefore, in this step, we independently analyze each file to extract the respective root cause deletion lines.

Current research demonstrates that code differs fundamentally from natural language [15], and the efficacy of applying closed-source LLMs in real-world security management scenarios remains questionable [16]. Therefore, to fully leverage the strengths of LLMs, we utilize natural language information from CVE descriptions to aid LLMs in understanding the code within vulnerability-fixing commits, thereby capturing the true causes of vulnerabilities.

Prompt: Identification of the Root Cause Deleted Line

Inputs Provided
1. CVE Descriptions
2. Diff of the bug-fixing commit.
3. A list of potential deleted lines
Your Task
1 Analyze the provided *CVE descriptions* and *Diff of the bug-fixing commit* to understand the nature of the bug.
2 I will provide a list of potential deleted lines. Please evaluate the likelihood of each line being the root cause of the bug.
3 Identify the line that is most likely to be the root cause of the bug and return its id.

```
## Output Format
{{
  "understand_of_CVE":"",
  "analysis": [
    {{
      "id": "",
      "assessment":
"[Impossible/Possible/Highly Probable]",
      "explanation": ""
    }}
  ],
  "likely_root_cause": {{
    "id": "",
    "explanation": "",
    "result": ""
  }}
}}
```

Fig. 7: Prompt for Step A

As shown in Fig. 6, three types of information form the basis for the input to LLM. *CVE description:* For each CVE, this paper retrieves the corresponding CVE description from the NVD. This message provides cues related to the vulnerability and is highly informative for understanding the vulnerability. *Candidate Line Numbers:* LLM-SZZ considers all deleted lines in the file and extracts the line numbers for each line, thus restricting the focus of the LLM to a specific range. *Diff Information:* Extract the hunk information for each deleted line to form a diff collection. The information provides context for the deleted lines while omitting irrelevant hunks, thereby reducing the overall volume of diff information and increasing the ratio of pertinent information.

When guiding LLM in task execution, we utilize prompt chaining to foster deeper thinking, where a task is split into subtasks to create a chain of prompt operations. Our prompt chain is illustrated in Fig. 7. Through this process, we ultimately obtain the root cause deletion line for each file.

Due to space limitations, we are unable to provide the complete prompt here. However, it has been included in the reproduction package[1].

B. Vulnerability Backtrace Decision

During the software version update process, the root cause line that carries the vulnerability may either remain unchanged or go through some modifications. Consequently, past research has explored the use of line mapping algorithm to match these changing deleted lines. This paper utilizes the Levenshtein distance [17] for initial selection, supplemented by the assistance of an LLM for secondary refinement. The Levenshtein distance is an edit distance that measures differences between two strings. The calculation formula can be represented as follows:

$$D[i,j]=\begin{cases}0 & \text{if } i=j=0\\ i & \text{if } j=0\\ j & \text{if } i=0\\ \min\begin{cases}D[i-1,j]+1\\ D[i,j-1]+1\\ D[i-1,j-1]+\text{cost}(s1[i],s2[j])\end{cases} & \text{otherwise}\end{cases}$$

$D[i,j]$ represents the minimum edit distance required to convert the first i characters of the first string into the first j characters of the second string, and $\text{cost}(s1[i], s2[j])$ represents the cost of replacing character $s1[i]$ with $s2[j]$, typically 0 or 1.

Prompt: Vulnerability Backtrace Decision

Inputs Provided
1. CVE Descriptions
2. Diff of the bug-fixing commit.
3. A list of potential deleted lines
Your Task
1 Analyze the provided *CVE descriptions* and *Diff of the bug-fixing commit* to understand the nature of the bug.
2 I will provide a list of potential deleted lines. Please evaluate the likelihood of each line being the root cause of the bug.
3 Return `-1` if none of the lines contains bug. Return the line number most likely to contain the bug, if such a line exists.

Fig. 8: Prompt for Step B

After identifying the root cause line A in step 1, the process continues to trace back more deeply from this starting point. The procedure begins with executing *git blame* to identify the previous commit. If this commit does not include any deleted lines, the traceback is terminated; otherwise, we further calculate the Levenshtein distance between each deleted line in the commit and line A. If the Levenshtein distance is less than 0.5, it indicates a significant difference from the root cause line, suggesting a lower likelihood of containing the bug, thus making it unsuitable for further traceback. We will discuss the determination of the Levenshtein distance later. This distance helps to exclude a portion of deleted lines likely unrelated to the bug. The remaining lines are considered valid candidates and submitted to the LLM. Unlike the previous use of LLM, this step grants the model the authority to autonomously stop the traceback. If the LLM determines that all candidate lines are unrelated to the bug based on the CVE description, the diff information of the vulnerability-fixing commit, the diff information of candidate lines, and the candidate line numbers, it will stop the traceback. Otherwise, it identifies the root cause line in that commit, which is the line most likely related to the vulnerability. In each iteration, LLM-SZZ selects at most one deleted line per file as the next root cause line for traceback. Once the traceback is stopped, LLM-SZZ identifies the earliest commit related to the bug as the vulnerability-inducing commit for that file.

In this task, the prompt we used is shown in Fig. 8. The output format of the LLM is similar to that of the prompt in Fig. 7.

C. Optimal Candidate Selection

A tangled commit refers to a commit that involves modifications across multiple files; consequently, it is necessary to exclude any files that are not relevant to the vulnerability.

Through the steps outlined above, each file modified in the vulnerability-fixing commit is associated with corresponding vulnerability-inducing commits. The final step is to use an LLM to identify the commit most closely related to the vulnerability, based on the CVE description and diff of the fixing commit.

For instance, consider CVE-2016-10049, as depicted in Fig. 1. The LLM-SZZ method tracks 6 deleted lines, identifying three possible vulnerability-inducing commits: *bf3c016*, *92a2c48* and *fcb94af*. Subsequently, the method filters out the irrelevant commit *92a2c48* and *fcb94af*, confirming *bf3c016* as the actual vulnerability-inducing commit.

In this task, we used the prompt shown in Fig. 9.

Prompt: Vulnerability Backtrace Decision

Inputs Provided
1. CVE Descriptions
2. Diff of the bug-fixing commit.
3. Lines from different files in descendant commits
Your Task
1 Analyze the provided *CVE descriptions* and *Diff of the bug-fixing commit* to understand the nature of the bug.
2 I will provide a list of potential deleted lines from different files. Please evaluate the likelihood of each line being the origin of the bug.
3 Return the line number most likely to contain the bug.

Fig. 9: Prompt for Step C

IV. Evaluation

We assess the effectiveness of our proposed method by addressing the following five research questions:

[1] https://github.com/juzizi44/LLM_SZZ

- **RQ1:** Which LLM is the best for LLM-SZZ?
- **RQ2:** How does the Levenshtein distance affect the experimental results?
- **RQ3:** How do the CVE description and the fixing-commit diff affect the experimental results?
- **RQ4:** How does LLM-SZZ perform in comparison to previous SZZ methods?
- **RQ5:** What is the performance of LLM-SZZ on deep vulnerabilities?

A. Datasets

The dataset used in this study comes from 172 manually verified vulnerabilities by Bao et al. [3], including 100 C/C++ and 72 Java cases. We performed manual validation through a three-person review process, resolving discrepancies via discussion. During this process, we found the Java project blynk-server was removed, two vulnerabilities lacked valid inducing commits. We corrected these issues in the dataset[2]. Consequently, the Java vulnerability count was adjusted to 69.

Detailed information on the dataset is presented in Table I.

TABLE I: Details of Datasets

Dataset	Vulnerability		Vulnerability-fixing Commit
	Shallow	Deep	
C/C++ Dataset	60	40	100
Java Dataset	25	44	82

In the dataset, 'Shallow' vulnerabilities refer to those for which the vulnerability-inducing commit can be identified with a single use of the blame command on the vulnerability-fixing commit. In contrast, 'Deep' vulnerabilities require multiple iterations of blame to trace back to the original vulnerability-inducing commit. 'Vulnerability-fixing Commit' denotes the number of commits that address these vulnerabilities. In Java datasets, a single vulnerability may be rectified through multiple fix commits, often resulting in the number of fix commits surpassing the number of identified vulnerabilities. Therefore, there are 82 fixing commits rather than 69 in Java datasets.

B. Baseline Methods

To assess the effectiveness of our method, we replicated the following five baseline methods for comparison:

B-SZZ [4] An original SZZ method that searches for vulnerability-fixing commits in the software's version history and traces back through these commits to identify the changes that introduced the bugs, thereby determining the origin of the vulnerabilities. Since the B-SZZ method used in this study is based on the git blame command, it implicitly utilizes the annotation graphs.

AG-SZZ [5] An enhanced SZZ method that utilizes the structure of annotation graphs to ignore non-semantic lines and formatting changes (such as indentation and bracket positioning).

[2]https://github.com/juzizi44/LLM_SZZ

MA-SZZ [6] An advanced version of AG-SZZ, this method excludes commits that do not involve source code changes, including branch changes, merges, and attribute modifications.

V-SZZ [3] A state-of-the-art SZZ method. By integrating line-mapping techniques, it enables multiple backward traceabilities, effectively tracing vulnerabilities back to earlier versions.

NEURAL-SZZ [9] The first method applies a heterogeneous graph attention neural network to locate the root cause line. To extract the vulnerability-inducing commits, NEURAL-SZZ needs to be used in combination with V-SZZ.

The implementations for B-SZZ, MA-SZZ, and RA-SZZ are derived from the replication package provided by [18], while V-SZZ is based on research by [3]. The open-source code for NEURAL-SZZ originates from [9]. We provide a package to reproduce our experiments, which is available at the following address: https://github.com/juzizi44/LLM_SZZ.

C. Evaluation Metrics

Consistent with previous studies [3, 19], we employed two widely recognized metrics, Recall and Precision, to evaluate the SZZ methods. They are calculated as follows:

$$R = \frac{|\text{correct}_c \cap \text{identified}_c|}{|\text{correct}_c|}$$

$$P = \frac{|\text{correct}_c \cap \text{identified}_c|}{|\text{identified}_c|}$$

$|\text{correct}_c|$ represents the total number of true vulnerability-inducing commits in the dataset, while $|\text{identified}_c|$ denotes the number of commits identified by the SZZ method as vulnerability-inducing commits.

In this study, we also calculated the F1-score:

$$F1 = 2 \times \frac{P \times R}{P + R}$$

It is noteworthy that upon careful verification, we discovered that V-SZZ [3] had implemented deduplication for $|\text{correct}_c|$ and $|\text{identified}_c|$. However, we found that in the dataset, different vulnerabilities sometimes share the same vulnerability-inducing commit; for example, the vulnerability-inducing commits of CVE-2009-1379, CVE-2009-1378, and CVE-2014-0221 are identical. Thus, deduplication could lead to inaccuracies in the results. Our evaluation took this into account and avoided this error.

D. Comparison of Using Different LLMs (RQ1)

To answer RQ1, we evaluated the performance of several different LLMs on the LLM-SZZ method. Specifically, we selected a range of representative open-source models, including DeepSeek-V2.5 [20] and Mixtral-8x7B-Instruct-v0.1 [21]. We also incorporated several closed-source models from OpenAI, such as GPT-4o mini [22], GPT-4o [23], GPT-3.5 Turbo [24], and OpenAI o1-mini [25]. For each model, we utilized the default parameters. The comparative results for these models are shown in Table II. In C/C++ datasets, GPT-3.5 Turbo

TABLE II: Results of LLM-SZZ using different LLMs

Dataset	LLM	Recall	Precision	F1-score
C/C++	GPT-4o mini	0.800	0.808	0.804
	GPT-4o	0.820	0.828	0.824
	GPT-3.5 Turbo	**0.830**	**0.838**	**0.834**
	DeepSeek-V2.5	0.810	0.818	0.814
	Mixtral-8x7B-Instruct-v0.1	0.800	0.808	0.804
	OpenAI o1-mini	0.820	0.828	0.824
Java	GPT-4o mini	0.750	0.718	0.734
	GPT-4o	0.765	0.732	0.748
	GPT-3.5 Turbo	**0.779**	**0.768**	**0.774**
	DeepSeek-V2.5	0.765	0.732	0.748
	Mixtral-8x7B-Instruct-v0.1	0.706	0.640	0.671
	OpenAI o1-mini	0.735	0.704	0.719

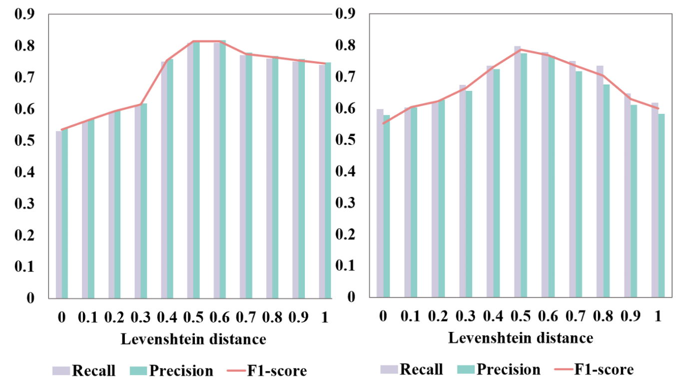
demonstrated superior performance, achieving a recall of 0.830, a precision of 0.838, and an F1 score of 0.834. In Java datasets, although the overall performance of GPT-3.5 Turbo was lower than in C/C++ datasets, it still performed the best among all models, with recall, precision, and F1 scores of 0.779, 0.768, and 0.774, respectively. **This suggests that the GPT-3.5 Turbo version of the LLM-SZZ method is most effective across different programming languages.** Therefore, the above experiment demonstrates the effectiveness of GPT-3.5 Turbo in this task. Taking cost considerations into account, DeepSeek-V2.5 has only a slight loss in performance compared to GPT-3.5 Turbo. However, with negligible expenses associated with this open-source model, DeepSeek-V2.5 demonstrates strong practical utility.

TABLE III: Stability Analysis of LLM-SZZ Using Different LLMs

Dataset	LLM	stable results	unstable results
C/C++	GPT-4o mini	0.905	0.041
	GPT-4o	0.845	0.042
	GPT-3.5 Turbo	0.859	0.081
	DeepSeek-V2.5	**0.960**	**0.000**
	Mixtral-8x7B-Instruct-v0.1	0.639	0.236
	OpenAI o1-mini	0.895	0
Java	GPT-4o mini	0.845	0.034
	GPT-4o	0.885	0.008
	GPT-3.5 Turbo	0.873	0.008
	DeepSeek-V2.5	**1.000**	**0.000**
	Mixtral-8x7B-Instruct-v0.1	0.686	0.212
	OpenAI o1-mini	0.869	0.025

To investigate the randomness of LLMs, we conducted a statistical analysis of the results obtained through the Voting Strategy. In each step of the individual requests, we considered the results stable if the outputs of the first three repeated runs of the model were identical, or if three out of the four outputs in the first four runs were the same. Conversely, we classified the results as unstable if, after five runs, the model produced identical outputs two times or fewer. The statistics for each model are presented in Table III. DeepSeek-V2.5 exhibited the highest stability, nearly reaching a value of 1. With the exception of Mixtral-8x7B-Instruct-v0.1, the majority of models demonstrated stability levels exceeding 84.5%, while completely random results were below 8.1%.

E. The impact of Levenshtein distance(RQ2)



(a) Performance comparison in C/C++ dataset (b) Performance comparison in Java dataset

Fig. 10: Performance comparison of different Levenshtein distances

Levenshtein distance can assist in filtering out lines that are not related to vulnerabilities before further analysis with an LLM. If the threshold is set too low, it may allow too much irrelevant information, which could interfere with the LLM's judgment. Conversely, if the threshold is set too high, it might inadvertently exclude lines that do contain vulnerabilities. To determine the appropriate Levenshtein distance, we conducted experiments incrementally increasing the distance from 0 by 0.1. We used GPT-3.5 Turbo as the LLM. The results of these experiments are shown in Fig. 10b and Fig. 10a. **The experiments indicated that the optimal performance was achieved when the Levenshtein distance was set to 0.5.**

F. The impact of CVE description and fixing-commit diff (RQ3)

To illustrate the significance of CVE descriptions and fixing-commit diffs in enabling LLMs to understand vulnerabilities during the Vulnerability Backtrace Decision process, we conducted ablation experiments for each component, as presented in Tables IV and V. **The data shows that removing either the CVE description or the fixing-commit diff leads to a decline in performance, with the fixing-commit diff having a greater effect.** This is because most CVE descriptions provide only a high-level overview of the vulnerability, often limited to function names, while fixing-commit diffs clearly outline specific code changes. As a result, code prompts allow LLMs to better grasp the intricate details of vulnerabilities, facilitating the backtracking process. However, when both components are provided, LLMs achieve a comprehensive understanding of vulnerabilities by integrating broader and more detailed insights, thereby resulting in enhanced performance.

G. Performance Comparison with Existing SZZs (RQ4)

To answer RQ4, we compared the performance of LLM-SZZ, which employs GPT-3.5 Turbo, against baseline methods in both C/C++ and Java datasets. The comparison results are

TABLE IV: Ablation Study on CVE Descriptions

Model	Dataset	Recall	Precision	F1-score
GPT-4o mini	Java	0.706	0.686	0.696
	C	0.780	0.788	0.784
	Average	0.743	0.737	0.740
GPT-4o	Java	0.735	0.714	0.725
	C	**0.800**	**0.808**	**0.804**
	Average	0.767	0.761	0.765
GPT-3.5 Turbo	Java	**0.750**	**0.739**	**0.745**
	C	**0.800**	**0.808**	**0.804**
	Average	**0.775**	**0.774**	**0.775**
DeepSeek-V2.5	Java	**0.750**	0.718	0.734
	C	**0.800**	**0.808**	**0.804**
	Average	**0.775**	0.763	0.769
Mixtral-8x7B-Instruct-v0.1	Java	0.632	0.589	0.610
	C	0.740	0.747	0.744
	Average	0.686	0.618	0.627
OpenAI o1-mini	Java	0.691	0.671	0.681
	C	0.780	0.788	0.784
	Average	0.735	0.730	0.733

TABLE V: Ablation Study on Fixing-Commit Diffs

Model	Language	Recall	Precision	F1-score
GPT-4o mini	Java	0.706	0.649	0.676
	C	0.740	0.747	0.744
	Average	0.723	0.698	0.710
GPT-4o	Java	0.721	0.690	0.705
	C	0.780	0.788	0.784
	Average	0.750	0.739	0.745
GPT-3.5 Turbo	Java	0.721	0.690	0.705
	C	**0.790**	**0.798**	**0.794**
	Average	0.756	0.744	0.750
DeepSeek-V2.5	Java	**0.735**	**0.704**	**0.719**
	C	0.780	0.788	0.784
	Average	**0.757**	**0.746**	**0.752**
Mixtral-8x7B-Instruct-v0.1	Java	0.706	0.600	0.649
	C	0.730	0.737	0.734
	Average	0.718	0.618	0.642
OpenAI o1-mini	Java	**0.735**	**0.704**	**0.719**
	C	0.760	0.768	0.764
	Average	0.748	0.736	0.742

presented in Table VI. Despite our best efforts to reproduce V-SZZ using the original dataset and code, the results still did not meet the performance claimed by the authors. Therefore, we use the results of our implementation.

First, we analyze the comparative experimental results within the C/C++ datasets. In terms of precision, LLM-SZZ performed the best, achieving a precision of 0.838, which is 18.8% higher than the best-performing baseline (V-SZZ at 0.650). For recall, LLM-SZZ led with a score of 0.830, surpassing V-SZZ by 3%. Regarding the F1-score, LLM-SZZ stood out with a score of 0.834, significantly exceeding V-SZZ (0.717) by approximately 11.7%.

In the evaluation experiments for the Java datasets, LLM-SZZ demonstrates a precision of 0.768, surpassing that of V-SZZ (0.504) by approximately 26.4% and NEURAL-SZZ (0.690) by 7.8%. Furthermore, the F1-score achieved by LLM-SZZ is 0.774, reflecting an improvement of about 14.4% over V-SZZ (0.630) and 8.8% over NEURAL-SZZ (0.686). The recall of LLM-SZZ (0.779) on the Java dataset exceeds that of NEURAL-SZZ(0.682) by 9.7%, but exhibits a decline of 6.2% relative to V-SZZ (0.841).

TABLE VI: Performance Comparison of Different SZZ Methods

Method	Dataset	Recall	Precision	F1-score
B-SZZ	C/C++	0.630	0.578	0.603
	Java	0.681	0.362	0.472
	Average	0.656	0.470	0.538
MA-SZZ	C/C++	0.660	0.520	0.581
	Java	0.783	0.432	0.557
	Average	0.722	0.476	0.569
AG-SZZ	C/C++	0.680	0.548	0.607
	Java	0.739	0.531	0.618
	Average	0.710	0.540	0.613
V-SZZ	C/C++	0.800	0.650	0.717
	Java	**0.841**	0.504	0.630
	Average	**0.821**	0.577	0.674
NEURAL-SZZ	Java	0.682	0.690	0.686
LLM-SZZ	C/C++	**0.830**	**0.838**	**0.834**
	Java	0.779	**0.768**	**0.774**
	Average	0.804	**0.803**	**0.806**

TABLE VII: Total Number of Vulnerability-inducing Commits Identified by SZZs

Method	Dataset	$\lvert \text{identified}_c \rvert$	False Positive
B-SZZ	C/C++	109	0.422
	Java	130	0.638
	Average	118.5	0.530
MA-SZZ	C/C++	127	0.480
	Java	125	0.568
	Average	126	0.524
AG-SZZ	C/C++	124	0.452
	Java	96	0.469
	Average	110	0.460
V-SZZ	C/C++	123	0.350
	Java	115	0.496
	Average	119	0.423
NEURAL-SZZ	Java	81	0.310
LLM-SZZ	C/C++	99	**0.162**
	Java	71	**0.232**
	Average	85	**0.197**

A vulnerability-fixing commit may involve modifications across multiple files, each containing numerous deleted lines; however, not all changes are directly related to the bug. Whether employing static methods or learning-based approaches, these techniques demonstrate a limited capacity for effectively filtering out noise. Consequently, even when they successfully identify vulnerability-inducing commits, the high false positive rates undermine the applicability of these methods in real-world scenarios.

Unlike these methods, LLM-SZZ was specifically designed to eliminate irrelevant candidate vulnerability-inducing commits from the outset. We analyzed the total number of vulnerability-inducing commits identified by various SZZ methods in C/C++ and Java datasets as detailed in Table VII. It is evident that the LLM-SZZ method identifies the fewest vulnerability-inducing commits across both programming languages, thus achieving the lowest false positive rate. In C/C++

datasets, LLM-SZZ identified only 99 commits, which is at least 10 fewer than other methods, with a false positive rate of only 0.162. Similarly, in Java datasets, LLM-SZZ reduced the count by more than 10 compared to its counterparts, with a false positive rate of 0.232. As the complexity of vulnerabilities increases and the number of modified files in vulnerability-fixing commits grows, the effectiveness of LLM-SZZ algorithm in reducing false positive rates becomes more significant.

Thus, we can conclude that LLM-SZZ exhibits the best overall performance on this dataset.

H. Deep Vulnerability Detection Analysis (RQ5)

To answer RQ5, we conducted a comparative analysis between the LLM-SZZ and V-SZZ methods to assess their recall in identifying deep and shallow software vulnerabilities. Due to the fact that methods like B-SZZ, MA-SZZ, and AG-SZZ only blame once to find the previous commit, they fail to identify deep vulnerabilities. Therefore, we do not discuss them in this part. We compiled comprehensive statistics from the experimental results, as summarized in Table VIII.

Unlike common programming errors, vulnerabilities typically originate in earlier software versions and undergo various degrees of code modifications during software updates. The line mapping algorithms approach in V-SZZ employs Levenshtein distance or Abstract Syntax Tree (AST) mapping algorithm. However, Levenshtein distance merely evaluates the similarity between lines, thus lacking depth. Additionally, AST mapping algorithm only considers the syntactic structure of the code, ignoring semantic details.

TABLE VIII: Comparison of Recall in Identifying Deep and Shallow Vulnerabilities Between LLM-SZZ and V-SZZ

Method	Language	Deep Vulnerabilities	Shallow Vulnerabilities
V-SZZ	C/C++	0.675	**0.867**
	Java	0.773	**0.880**
LLM-SZZ	C/C++	**0.900**	0.767
	Java	**0.787**	0.764

LLM-SZZ enhances the analysis by utilizing CVE description and natural language processing to aid in understanding the semantic aspects of vulnerabilities. This method not only assesses the similarity of deleted lines between consecutive commits but also leverages additional natural language information to grasp the underlying reasons for vulnerabilities. **Consequently, it performs more adeptly in tracing deep vulnerabilities by using both line similarity and natural language insights for decision-making.** However, LLM-SZZ does not outperform V-SZZ in detecting shallow vulnerabilities, as it may over-trace back to commits before the introduction of the vulnerabilities.

I. Limitations

We conducted an analysis of the error causes for each vulnerability-fixing commit by the LLM-SZZ and V-SZZ, categorizing the errors into three types: 1) Insufficient backtracking, where the actual vulnerability-introducing commit occurred earlier than the commit identified by SZZ; 2) Excessive backtracking, where the actual vulnerability-inducing commit is more recent than the commit identified by SZZ; 3) Misidentification, where no vulnerability-introducing commits were identified during the backtracking process, or an incorrect backtracking path was chosen.

TABLE IX: Analysis of Error Causes in LLM-SZZ and V-SZZ Methods

Method	Error Types		
	Insufficient Backtracking	Excessive Backtracking	Misidentification
V-SZZ	17	10	4
LLM-SZZ	8	14	10

As indicated in Table IX, we observed that V-SZZ primarily suffers from insufficient backtracking, whereas LLM-SZZ is more prone to excessive backtracking. This pattern suggests that LLM-SZZ tends to backtrack more deeply, erroneously identifying non-buggy lines as buggy. This issue arises because the model may consider a line to have evolved from a line in an earlier commit during backtracking, despite the line in that earlier commit being bug-free, thus prompting further unnecessary backtracking.

J. Case Study: Why GPT-4o Fails

Since GPT-3.5 Turbo outperformed GPT-4o, which is counterintuitive, we specifically selected a vulnerability case for analysis. Using CVE-2016-2182 [26] in the OpenSSL project as an example, GPT-3.5 Turbo successfully identified the vulnerability, whereas GPT-4o failed to analyze it correctly. The model outputs are presented in Table X.

We found that both models effectively understood the vulnerability and demonstrated that GPT-4o provides a more comprehensive and detailed analysis of the code. However, discrepancies arose during the second step of analyzing each candidate line. GPT-3.5 Turbo considered the change in the line involving memory allocation for the 'bn_data' array within the 'BN_bn2dec()' function, thus linking this line to the vulnerability; while GPT-4o determined that the change in line 118 merely involved removing typecasting from the allocation of memory to 'bn_data' and 'buf'. This modification does not directly relate to checking the return values of mathematical operations or managing buffer boundaries, which were the core issues in Bug A. Both GPT-4o and GPT-3.5 Turbo show strong reasoning, but GPT-4o's stricter logic may miss weak or implicit signals, leading to lower task performance. GPT-3.5 Turbo, being more lenient, better identifies "potentially relevant" cues. This is a task-specific limitation of GPT-4o, not a general flaw.

V. Threats to Validity

Construct Validity. An issue related to construct validity arises from our use of the dataset provided by Baolingfeng

TABLE X: Comparison of GPT-3.5 Turbo and GPT-4o in Analyzing CVE-2016-2182

Output	GPT-3.5 Turbo (True)	GPT-4o (False)
Description of Vulnerability	Bug A in the CVE description involves a vulnerability in the 'BN_bn2dec()' function of OpenSSL. When an oversized BIGNUM is passed, it can trigger a failure in 'BN_div_word()', leading to out-of-bounds writes to the 'bn_data' buffer, potentially causing a crash. The fix checks the return value of 'BN_div_word()' to prevent overflow.	The issue arises when handling large BIGNUMs in 'BN_bn2dec()', which converts BIGNUMs to decimal strings. If an extremely large BIGNUM is passed, division using 'BN_div_word(t, BN_DEC_CONV)' might fail (returning -1). This failure wasn't checked in earlier versions, causing out-of-bounds writes as the pointer 'lp' would increment regardless of success. The fix ensures pointer 'lp' increments don't exceed the allocated size of 'bn_data'.
Analysis of Each Candidate Line	Line 118: Highly Probable - The change in this line involves memory allocation for the 'bn_data' array within the 'BN_bn2dec()' function, addressing the potential buffer overflow issue discussed in the bug description.	Line 118: Impossible - The change in line 118 deals only with removing typecasting from the allocation of memory to 'bn_data' and 'buf'. **This modification does not directly relate to checking the return values of mathematical operations or managing buffer boundaries, which were the core issues in Bug A.** Hence, there is no likelihood that Bug A resided in this particular line in its previous version.
Result	118	-1

et al. This dataset serves as a foundational element for our analysis, and its correctness is critical for drawing accurate conclusions. To ensure the accuracy of the true inducing commits, we undertook a meticulous process of manual verification for all vulnerabilities. Specifically, we employed a three-person review mechanism, where each case was independently examined and any disagreements were resolved through group discussion. The finalized, corrected dataset has been made publicly available in our replication repository for transparency and reproducibility.

Internal Validity. One assumption of our method is that statements directly related to vulnerabilities are deleted while those addressing them are added. However, this assumption may not hold true in certain situations, such as cases where there are no deleted lines. Nonetheless, all SZZ algorithms are based on this assumption, and our research aims to enhance the precision of SZZ methods. Additionally, an important consideration for internal validity is the inherent randomness of large language models (LLMs). To evaluate the impact of this randomness on our task, we analyzed the proportion of unstable results and found that, for certain models, this randomness rendered the results unreliable. Therefore, it is advisable to avoid using LLMs with excessive randomness for this task.

External Validity A potential threat to the external validity of our study is the total number of vulnerabilities, which stands at 172. This figure includes both deep and shallow vulnerabilities and is comparable to the number of bugs analyzed in previous studies [3, 6, 27, 28]. This suggests that the scale of our dataset is consistent with established research in the field.

VI. Related Work

In this chapter, we present LLMs and their applications in the field of security; then we review various variants of the SZZ method.

A. Large Language Model

LLM consists of billions of parameters, primarily based on the Transformer architecture [29], which includes multiple self-attention layers. The model is capable of focusing on different parts of the input data and deeply learning the semantics and grammatical structures of the language. Currently, various LLMs have emerged [30, 31]. Developed from DeepSeek LLM [32], DeepSeek-V2 [33] introduced the Multi-head Latent Attention (MLA) structure and utilized the proprietary DeepSeekMoE technology to further reduce computational load, significantly enhancing inference efficiency. Recently, DeepSeek-V2.5 has emerged as an integration of DeepSeek-V2-Chat and DeepSeek-Coder-V2-Instruct, thereby consolidating the general and coding capabilities of its predecessors [20]. Besides, Albert Q. Jiang et al. introduced Mixtral 8x7B [21], a Sparse Mixture of Experts (SMoE) language model. StarCoder2 [34] relies on the digital commons provided by the Software Heritage (SWH) source code archive for its foundation. The development of GPT-3.5 [24] mainly followed the training methods of InstructGPT, with targeted optimizations for conversational capabilities. GPT-4 [35], for the first time, expanded the input modalities of the GPT series models from single-text to both text and image. Furthermore, OpenAI o1 [36] employs a chain of thought processes to enhance its reasoning capabilities through reinforcement learning, enabling it to recognize and rectify mistakes, simplify complex steps, and adapt its strategies when confronted with challenges.

LLMs are also widely applied in the field of security. In static analysis, Li Haonan et al. [37] explores the potential roles of LLMs in this area by posing relevant questions and assessing the practicality of employing LLMs for program analysis. Llm4sa [38] can remove a great deal of false warnings and facilitate bug discovery significantly. Additionally, The team led by Li introduced LLift [39], an innovative framework that merges static analysis with LLMs, focusing on the detection of use-before-initialization (UBI) bugs in the Linux kernel. LLMs have also demonstrated excellent performance in fuzz testing within the dynamic analysis field. Fuzz4all [40] leverages LLMs as an input generation and mutation engine, which enables the approach to produce diverse and realistic inputs for any practically relevant language. BusyBox [41] employs LLMs to generate target-specific initial seeds. Moreover, Huang Linghan et al. [42] have developed LATTE, which is the first static binary taint analysis supported by an LLM.

B. SZZ Methods

Several SZZ methods have been proposed to identify vulnerability-inducing commits. The B-SZZ method [4] was originally proposed by Śliwerski in 2005. It straightforwardly blames vulnerability-fixing commits, determines the earlier change at the location of the fix, and then considers all the resulting candidates as vulnerability-inducing commits. Although effective for straightforward bugs, it performs poorly for complex bug situations.

AG-SZZ [5] aims to improve upon the original B-SZZ by excluding non-semantic lines and changes in code formatting, such as indentation and bracket adjustments, which often led to processing a vast amount of irrelevant modifications. By incorporating an annotation graph structure, AG-SZZ effectively filters out whitespace and comments, significantly reducing false positives. MA-SZZ [6] was designed by Da Costa et al. It focuses on excluding meta-changes, which are those commits that do not involve source code modifications, such as branch changes and file attribute modifications, from being considered as potential vulnerability-fixing commits. Unlike AG-SZZ, which mistakenly marks these meta-changes as vulnerability-inducing commit candidates when using the annotation graph, MA-SZZ narrows down the candidates by ruling out all commits without source code changes.

Furthermore, V-SZZ [3] emphasizes the identification of vulnerabilities that may have been introduced in earlier versions of software. Traditional SZZ methods typically focus only on the most recent modifications and are ineffective in tracing back to vulnerabilities that may exist in earlier versions. V-SZZ leverages line mapping techniques for multiple reverse tracing iterations. For Java datasets, it employs an AST mapping algorithm [43]; for C/C++ datasets, it utilizes Levenshtein distance [17] to map similar lines. This method significantly enhances the understanding of the origins of vulnerabilities, especially those introduced early on and persisting over time. Besides, to identify the root cause line of vulnerability-fixing commits, NEURAL-SZZ [9] employs a Heterogeneous Graph Attention Network (HAN) to capture the semantic relationships between lines. Tang et al. propose using NEURAL-SZZ initially, followed by the application of V-SZZ for tracing back, which significantly improves precision. However, this approach is limited to the Java programming language and has a high false positive rate.

VII. Conclusion and Future Work

The existing SZZ method is plagued by a high rate of false positives. To address this issue, we propose the high-precision LLM-SZZ method.

Our approach innovatively utilizes Large Language Models (LLMs) to improve the accuracy of identifying vulnerability-inducing commits while substantially reducing false positives. To address the challenges associated with inaccurately identifying root cause lines in iterative commits, we employed an LLM that incorporates natural language information from CVE descriptions alongside vulnerability-fixing commit diffs to better understand code semantics. To tackle the insufficient backtracking capability in line matching, we integrated a line mapping algorithm with LLM analysis and designed a prompt chain for the LLM. For evaluation, we utilized a real-world dataset that is widely used in assessing SZZ methods.

Experimental results demonstrate that the LLM-SZZ method, leveraging the GPT-3.5 Turbo model, achieved precision improvements of 18.8% and 26.4% and F1 score enhancements of 11.7% and 14.4% in vulnerability tracing tasks for C/C++ and Java datasets, respectively, compared to the current state-of-the-art method. These significant advantages not only confirm the enhanced effectiveness of LLM-SZZ over traditional SZZ approaches but also highlight its efficiency and reliability in handling complex vulnerability scenarios.

In the future, we plan to create a dataset encompassing a wider range of programming languages to evaluate the effectiveness of LLM-SZZ on additional data. Additionally, we intend to explore its potential application in common bugs beyond vulnerabilities, thereby expanding the scope of utility for LLM-SZZ.

References

[1] Apache, "Cve-2021-44228," 2021. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2021-44228

[2] C. S. R. Board, "Csrb report on log4j," Jul 2022.

[3] L. Bao, X. Xia, A. E. Hassan, and X. Yang, "V-szz: automatic identification of version ranges affected by cve vulnerabilities," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2352–2364.

[4] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[5] S. Kim, T. Zimmermann, K. Pan, E. James Jr *et al.*, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM international conference on automated software engineering (ASE'06)*. IEEE, 2006, pp. 81–90.

[6] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2016.

[7] E. C. Neto, D. A. Da Costa, and U. Kulesza, "The impact of refactoring changes on the szz algorithm: An empirical study," in *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2018, pp. 380–390.

[8] P. Rani, F. Petrulio, and A. Bacchelli, "On refining the szz algorithm with bug discussion data," *Empirical Software Engineering*, vol. 29, no. 5, p. 115, 2024.

[9] L. Tang, L. Bao, X. Xia, and Z. Huang, "Neural szz algorithm," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1024–1035.

[10] K. Herzig, S. Just, and A. Zeller, "The impact of tangled code changes on defect prediction models," *Empirical Software Engineering*, vol. 21, pp. 303–336, 2016.

[11] MITRE, "Cve-2017-1000398," 2018. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2017-1000398

[12] mitre, "Cve-2016-10049," 2017. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2016-10049

[13] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[14] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," *arXiv preprint arXiv:2303.12712*, 2023.

[15] H. He, X. Lin, Z. Weng, R. Zhao, S. Gan, L. Chen, Y. Ji, J. Wang, and Z. Xue, "Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection," in *33rd USENIX Security Symposium (USENIX Security 24), PHILADELPHIA, PA*, 2024.

[16] X. Liu, Y. Tan, Z. Xiao, J. Zhuge, and R. Zhou, "Not the end of story: An evaluation of chatgpt-driven vulnerability description mappings," in *Findings of the Association for Computational Linguistics: ACL 2023*, 2023, pp. 3724–3731.

[17] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[18] G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto, "Evaluating szz implementations through a developer-informed oracle," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 436–447.

[19] E. C. Neto, D. A. Da Costa, and U. Kulesza, "Revisiting and improving szz implementations," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–12.

[20] DeepSeek, "Deepseek-v2.5," 2024. [Online]. Available: https://huggingface.co/deepseek-ai/DeepSeek-V2.5

[21] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. de las Casas, E. B. Hanna, F. Bressand, G. Lengyel, G. Bour, G. Lample, L. R. Lavaud, L. Saulnier, M.-A. Lachaux, P. Stock, S. Subramanian, S. Yang, S. Antoniak, T. L. Scao, T. Gervet, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mixtral of experts," 2024.

[22] OpenAI, "Gpt-4o mini," 2024. [Online]. Available: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

[23] ——, "Gpt-4o," 2024. [Online]. Available: https://openai.com/index/hello-gpt-4o/

[24] ——, "Introducing chatgpt," 2022. [Online]. Available: https://openai.com/blog/chatgpt

[25] ——, "Openai o1 mini," 2024. [Online]. Available: https://openai.com/index/openai-o1-mini-advancing-cost-efficient-reasoning/

[26] H. Red, "Cve-2016-2182," 2016. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2016-2182

[27] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona, "How bugs are born: a model to identify how bugs are introduced in software components," *Empirical Software Engineering*, vol. 25, pp. 1294–1340, 2020.

[28] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us? a taxonomical study of large commits," in *Proceedings of the 2008 international working conference on Mining software repositories*, 2008, pp. 99–108.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[30] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang *et al.*, "A survey on evaluation of large language models," *ACM Transactions on Intelligent Systems and Technology*, vol. 15, no. 3, pp. 1–45, 2024.

[31] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[32] X. Bi, D. Chen, G. Chen, S. Chen, D. Dai, C. Deng, H. Ding, K. Dong, Q. Du, Z. Fu *et al.*, "Deepseek llm: Scaling open-source language models with longtermism," *arXiv preprint arXiv:2401.02954*, 2024.

[33] DeepSeek-AI, "Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model," 2024.

[34] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder 2 and the stack v2: The next generation," 2024.

[35] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[36] OpenAI, 2024. [Online]. Available: https://openai.com/index/hello-gpt-4o

[37] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Assisting static analysis with large language models: A chatgpt experiment," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 2107–

2111.

[38] C. Wen, Y. Cai, B. Zhang, J. Su, Z. Xu, D. Liu, S. Qin, Z. Ming, and C. Tian, "Automatically inspecting thousands of static bug warnings with large language model: How far are we?" *ACM Transactions on Knowledge Discovery from Data*, 2024.

[39] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.

[40] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[41] Y. Oliinyk, M. Scott, R. Tsang, C. Fang, H. Homayoun *et al.*, "Fuzzing busybox: Leveraging llm and crash reuse for embedded bug unearthing," *arXiv preprint arXiv:2403.03897*, 2024.

[42] L. Huang, P. Zhao, H. Chen, and L. Ma, "Large language models based fuzzing techniques: A survey," *arXiv preprint arXiv:2402.00350*, 2024.

[43] Y. Fan, X. Xia, D. Lo, A. E. Hassan, Y. Wang, and S. Li, "A differential testing approach for evaluating abstract syntax tree mapping algorithms," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1174–1185.