Code Change Intention, Development Artifact and History Vulnerability: Putting Them Together for Vulnerability Fix Detection by LLM

ANONYMOUS AUTHOR(S)

Detecting vulnerability fix commits in open-source software is crucial for maintaining software security. To help OSS identify vulnerability fix commits, several automated approaches are developed. However, existing approaches like VulFixMiner and CoLeFunDa, focus solely on code changes, neglecting essential context from development artifacts. Tools like Vulcurator, which integrates issue reports, fail to leverage semantic associations between different development artifacts (e.g., pull requests and history vulnerability fixes). Moreover, they miss vulnerability fixes in tangled commits and lack explanations, limiting practical use. Hence to address those limitations, we propose LLM4VFD, a novel framework that leverages Large Language Models (LLMs) enhanced with Chain-of-Thought reasoning and In-Context Learning to improve the accuracy of vulnerability fix detection. LLM4VFD comprises three components: (1) Code Change Intention, which analyzes commit summaries, purposes, and implications using Chain-of-Thought reasoning; (2) Development Artifact, which incorporates context from related issue reports and pull requests; (3) Historical Vulnerability, which retrieves similar past vulnerability fixes to enrich context. More importantly, on top of the prediction, LLM4VFD also provides a detailed analysis and explanation to help security experts understand the rationale behind the decision. We evaluated LLM4VFD against state-of-the-art techniques, including Pre-trained Language Model-based approaches and vanilla LLMs, using a newly collected dataset, BigVulFixes. Experimental results demonstrate that LLM4VFD significantly outperforms the best-performed existing approach by 68.1%–145.4%. Furthermore, We conducted a user study with security experts, showing that the analysis generated by LLM4VFD improves the efficiency of vulnerability fix identification.

CCS Concepts: • **Software and its engineering**;

Additional Key Words and Phrases: Vulnerability Fix Detection, Large Language Model

ACM Reference Format:

Anonymous Author(s). 2025. Code Change Intention, Development Artifact and History Vulnerability: Putting Them Together for Vulnerability Fix Detection by LLM. 1, 1 (January 2025), 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software development heavily relies on the use of open-source software (OSS). However, failing to detect and mitigate vulnerabilities in OSS can lead to catastrophic consequences [62]. OSS organizations generally follow the Coordinated Vulnerability Disclosure (CVD) [49] model to disclose vulnerabilities. In this model, details of vulnerabilities are publicly shared once the developers feel they have had sufficient time for remediation of the security risk. This often causes a delay between when a commit that fixes a vulnerability (*i.e., vulnerability fix commit*) is integrated into the codebase and when the vulnerability and or its fix are publicly announced. The time between fix and disclosure can provide malicious users a window of opportunity to find details of vulnerabilities and exploit them in software dependent on the OSS. Although CVD recommends applying fixes silently

ACM XXXX-XXXX/2025/1-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

to avoid leaking sensitive information about the vulnerability, the risk of exploitation remains. Hence, OSS users have a crucial incentive to monitor integrated OSS and discover vulnerability fixes to begin the remediation process as fast as possible (*i.e., vulnerability fix detection*).

Vulnerability fix detection is a complex task and poses a significant challenge for general OSS organizations without an automated approach. Furthermore, manually monitoring all commits across integrated OSS is highly time-consuming and costly. To solve this, automated approaches to identify vulnerability fixes have been proposed in prior studies, which typically train a deep learning model using commit-level information. For instance, *VulFixMiner* [72] trains a pre-trained language model (PLM) using code change information from commits. In their next work [71], they proposed an approach to detect vulnerability fixes at a finer granularity by using function-level code change information. However, those approaches only leverage code change information and neglect additional software development artifacts related to commits, making it challenging to identify the intent of code changes without sufficient context, typically for nuanced code changes (*e.g.,* adding condition check). Vulnerability fix is a complex task that is often associated with issue reports [52], but such information is not adequately utilized in existing methods [71, 72]. The only work that leverages development artifacts (*i.e.,* issue report) is Vulcurator [39], however, it used a voting mechanism without leveraging the semantic association between artifacts. Previous approaches also fail to identify vulnerability fixes within a tangled commit — those with a mixture of code modifications for various purposes (*e.g.,* refactoring) [4]. The limitations of existing approaches mentioned above cause them to miss many true vulnerability fix commits (evidenced by very low recalls (0.06 to 0.26) as shown in Section 6.1). More importantly, identifying vulnerability fixes requires specialized security expertise and project-specific domain knowledge [36, 75], prior approaches only provide a prediction, without an explanation behind this decision, which hinders the use of such approaches in practice.

Recently, Large Language Models (LLMs) have demonstrated promising results in code-related tasks, such as code understanding [2, 32] and vulnerability understanding [18, 27]. Intuitively, vulnerability fix detection is a task that requires the abilities of natural language understanding and code comprehension, typically in vulnerability understanding. Hence, the strong natural language and code understanding capabilities of LLMs fit the requirements of this task well.

Therefore, to tackle the problem and overcome existing limitations (*i.e.,* neglecting information in development artifacts, tangled commits, and lack of sufficient explanation) of previous works, we propose *LLM4VFD*, a novel framework that enhances vulnerability fix detection by leveraging multiple sources of information distilled by LLMs, consisting of four components. The Code Change Intention (CCI) component analyzes a commit to extract its summary, purpose, and implications, using Chain-of-Thought (CoT) reasoning. The Development Artifact (DA) component utilizes information from related development artifacts, such as issue reports (IRs) and pull requests (PRs), to provide additional context and enhance understanding of the commit. The Historical Vulnerability (HV) component retrieves similar vulnerability fix commits from historical vulnerability data to further enhance the context of the commit. Last, the Comprehensive Analysis and Vulnerability Fix Detection (CAVFD) component combines the information distilled from the previous components to enable In-Context Learning. The synthesized information is used to generate a final prediction, while also providing a detailed explanation in analysis to help security experts understand the rationale behind the decision.

We evaluated LLM4VFD on a newly created multi-language vulnerability fix dataset which we call the BigVulFix dataset. This dataset consists of 1,689 vulnerability fix commits after 2023 collected from the National Vulnerability Database (NVD) [43]. LLM4VFD is evaluated on 6 LLMs spanning 3 LLM families in varying parameter sizes (*i.e.,* 7B–236B). LLM4VFD outperforms PLM-based approach consistently in terms of MCC, F1-score, and recall. For instance, LLM4VFD outperforms the best-performed PLM-based approach VulCurator by 68.1%–145.4% in terms of F1-score when using different LLMs as the base model. Our framework demonstrates performance gains ranging from 12.7%–105.6% over its vanilla variant, with smaller models generally benefiting more compared to their larger counterparts. In addition, the conducted ablation analysis shows that all three components are crucial in contributing to the final performance of LLM4VFD. Through a user study involving security experts, we find in 80.0% of the cases LLM4VFD's analysis helps security experts to understand the code change and improve the efficiency in identifying whether the change is a vulnerability fix commit. We also conducted a failure case analysis to understand the limitations of LLM4VFD, and provide insights for future work.

In summary, this paper makes the following contributions:

- To the best of our knowledge, we are the first study to investigate vulnerability fix detection using LLM.
- We propose a novel framework, LLM4VFD, that enhances vulnerability fix detection by leveraging multiple sources of information based on LLMs.
- We conducted extensive evaluation on LLM4VFD, including an ablation study to evaluate the contribution of each component in LLM4VFD and a user study to evaluate the usefulness of LLM4VFD's analysis result for security experts.
- We collected and release a new dataset, BigVulFixes, containing 1,689 vulnerability fix commits from 7 major programming languages after 2023 to avoid data leakage with LLM's pre-training data.
- We perform a bad case analysis on the limitations of our approach to promote future work on vulnerability fix detection.

2 BACKGROUND AND RELATED WORK

In this section we introduce the background of vulnerability fix detection and LLMs.

2.1 Vulnerability Fix Detection

A typical vulnerability fix detection task takes a commit as input and outputs whether the commit is fixing a vulnerability. Existing approaches focusing on this problem mostly leverage PLM techniques through some kind of embedding technique to capture code change information. The first work on vulnerability fix detection is introduced by Zhou and Sharma [76]. They used commit messages and bug reports to automatically identify vulnerability fix commits. Chen et al. [9] later explored the topic with the idea of vulnerability-relatedness, to capture how each commit can be related to addressing a vulnerability.

During this time, trying to associate a commit with a vulnerability was mainly to curate vulnerability-related information to aid data collection and cleaning. Having high-quality data for vulnerability fixing code is critical for other vulnerability tasks such as automated vulnerability repair. Zhou et al. [72] further explored the idea of detecting vulnerability fix commit especially in the case of silent fixes where the vulnerability fix information may not be publicly disclosed. Sabetta and Bezzi [58] leveraged an SVM based approach with features from commit messages and patch information to predict whether a commit is security related. Nguyen-Truong et al. [42] combined three classifiers each for the patch, commit, and associated issue content to create a joint classifier to identify vulnerability fix commits. Xu et al. [66] proposed SPAIN to identify vulnerability fixes at the binary level. Their work is especially useful when the target software is not available in source code and is distributed only in binary.

There have also been efforts in the direction of a more fine-grained level of vulnerability detection. For example, Zhou et al. [71] proposed a framework to analyze vulnerable code at the function level using deep learning techniques. Their framework can identify vulnerability fixes, CWE category, and exploitability. Alternative representations of the changed code have also been explored, especially using graph-based techniques. Nguyen et al. [39] proposed a graph-based approach for detecting silent vulnerability fixes. They construct graphs using the AST before and after a commit. Their work show significant improvement in performance in real-world C/C++ projects.

2.2 Large Language Models

Popularized by Generative Pretrained Transformer (GPT) models [50], LLMs have shown great potential in software engineering tasks. Recent work explores the capabilities of LLMs in solving several unique software engineering problems [10, 17, 26, 27, 30, 74]. In practice, LLMs are generally tailored for use in domain-specific tasks through prompt engineering and/or fine-tuning.

Prompt engineering is an essential step in interacting with an LLM to alter its response. The characteristics of the prompt (*e.g.,* vocabulary, style, tone) can greatly affect the generated response of the LLM [69]. Carefully crafted prompts can help improve the capability of LLMs in specific tasks. For example, CoT [64] is a common form of prompt engineering that separates the prompt into smaller, individual steps, which improves the reasoning capabilities of LLMs. LLMs also have varying context lengths which limit the length of the prompt, ultimately truncating information that exceeds this limit. Thus, prompt engineering also generally aims to tailor prompts to become brief by condensing information. In addition, prompt engineering is an effective way to work around the knowledge cutoff of LLMs. LLMs are typically trained with information that precedes a certain knowledge cutoff date. When LLMs are queried for information beyond the cutoff date or not in the training dataset, they often hallucinate and create unsuitable answers. An example of this often arises from 0-shot prompting strategies [29], which prompt LLMs to perform tasks that they are not explicitly trained in. This issue is often resolved by providing contextual information in the prompt. A canonical solution is In-Context Learning (ICL) [13], which includes examples and/or demonstrations of the task in the prompt.

With its superior capability in code related tasks such as code understanding [59], code summary [63], and code generation [3, 19, 28, 35]), LLM overcomes many limitations of previous techniques [26, 65]. The strength of LLMs leads to researchers adopting the tool for vulnerability related tasks. Most of the previous work has focused on vulnerability detection [73]. Chan et al. [6] proposed a system based on vulnerable code patterns to detect vulnerabilities in code. Thapa et al. [61] researched LLM's performance of C/C++ source codes with multiple vulnerabilities. Cheng et al. [10] presented an approach Vercation to identify vulnerable versions of open-source C/C++ software. Du et al. [14] proposed Vul-RAG, a framework that first constructs a vulnerability knowledge base that contains CVE information. The framework is then able to predict whether a given code snippet is vulnerable by an RAG system that retrieves from the knowledge base.

3 CHALLENGES AND MOTIVATION

In this section, we outline the primary challenges faced by current vulnerability fix detection techniques and discuss the types of context information that could be leveraged to address them.

3.1 Challenge #1: Tangled Commits

One of the key challenges in detecting vulnerability fixes is identifying security-related changes within a commit that contains multiple modifications for various purposes, such as feature improvements, refactoring, and vulnerability resolution [4]. This mix of intentions complicates the task for traditional methods [25], particularly those that rely solely on analyzing code changes to detect vulnerabilities [71, 72]. For example, in Figure 1, only two lines out of 164 in the commit address a vulnerability by altering the initialization and usage of `LookupPathMatchableHandlerMapping`,

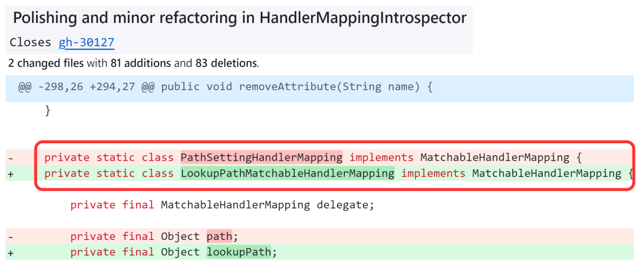

Fig. 1. An example of tangled commit with 164 lines changed, while only two lines (in red box) are related to vulnerability fix. [60]

while the rest relates to refactoring and feature adjustments. The substantial amount of non-vulnerability-related changes in such tangled commits introduces noise, making it difficult for existing approaches to accurately detect the real vulnerability-related fixes. In fact, none of the current methods [39, 71, 72] can identify this particular case.

To tackle this issue, we interviewed security experts to understand how they discern vulnerability fixes, especially in tangled commits. From their insights, we identified three crucial aspects of information needed to determine a vulnerability fix: the summary of the code change, the purpose behind the changes, and their potential implications. By concentrating on these aspects rather than just the raw code changes, we can effectively filter out noise from unrelated modifications, thereby enhancing the accuracy of models for vulnerability fix detection.

3.2 Challenge #2: Insufficient Information Within Commits

In many cases, commit messages and code changes alone do not provide enough information to determine whether a commit is related to a vulnerability fix. This challenge is particularly pronounced when changes are subtle and or the commit message is vague, making it difficult for traditional methods [71, 72] to accurately identify vulnerability fix commits. For example, the commit shown in Figure 2 simply adds three lines of code to `filter_session.c` with a conditional check. The commit message, "fixed #2475," provides little insight into the nature of the changes or whether they are security relevant. Hence, based on the code and commit message alone it is nearly impossible to identify whether this is a vulnerability fix or a routine update, even for human experts. However, when we investigate the related development artifact — issue report #2475 — it becomes evident that this commit addresses a security vulnerability. The issue report details a vulnerability involving improper handling of certain filter parameters, which could lead to a security flaw causing an out-of-bounds read and segmentation fault. The commit directly resolves this issue, though without the context provided by the issue report, commit-only methods are likely to miss this crucial connection and result in a false negative prediction. Therefore, we aim to leverage related development artifacts to enrich commits for vulnerability fix detection.

3.3 Challenge #3: Missing Project-Specific Contextual Information

It is usually challenging to predict only based on the code changes without a deeper understanding of the corresponding OSS software (i.e., project-specific domain knowledge). For instance, a commit shown on the left in Figure 3 includes a small code change that adds checks in the `setProperty` function to prevent changes to the prototype property — a common source of prototype pollution vulnerabilities. While the commit message uses the word "fix", the simplicity of the change makes it difficult to determine whether this is a vulnerability fix, typically for traditional methods, which

Fig. 2. An example of a commit [21] with only 2 lines changed (in red box), while related issues reports [22] (in blue box) provided critical information.

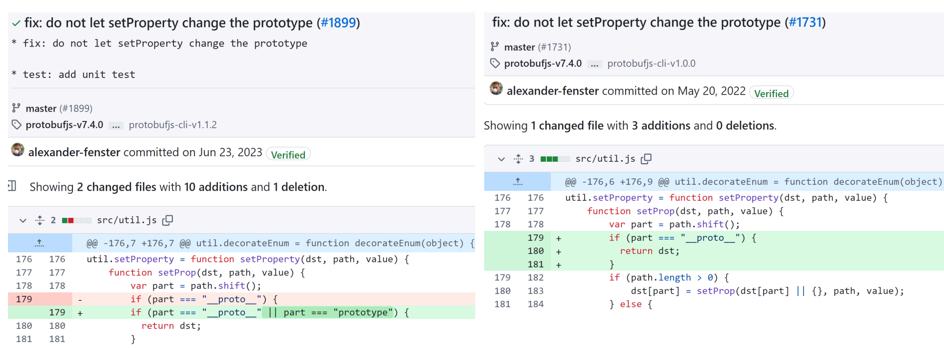

Fig. 3. A commit only with small change by adding condition check (left [55]) and its relevant historical vulnerability fix commit (right [54]).

rely on a surface-level analysis of the commit message and or code [39, 71, 72], may struggle to differentiate between them without such information. Conveniently, projects typically have historical commits that possibly contain similar fixes can provide more context-specific for the project. For instance, in Figure 3 (right), we present a historical vulnerability fixing commit, which includes similar changes to the same function in a prior commit, where the addition of a check for the `__proto__` property was introduced to mitigate a known prototype pollution vulnerability. This example shows that historical vulnerability fixes can be leveraged to supply the missing context needed to identify new vulnerability fix commits.

4 METHODOLOGY

To address the challenges discussed in Section 3, we present LLM4VFD, which leverages information that is distilled from multiple sources by leveraging LLMs to enhance the vulnerability fix detection. Figure 4 provides an overview of our framework. LLM4VFD consists of four components: Code Change Intention (CCI), Development Artifact (DA), Historical Vulnerability (HV), and Comprehensive Analysis and Vulnerability Fix Detection (CAVFD).

More specifically, to address Challenge #1, the CCI component leverages LLMs with Chain-of-Thought (CoT) techniques to construct a summary of the commit that focuses on three crucial aspects (*i.e.*, code change summary, purpose of the change, and implications of the change). To address

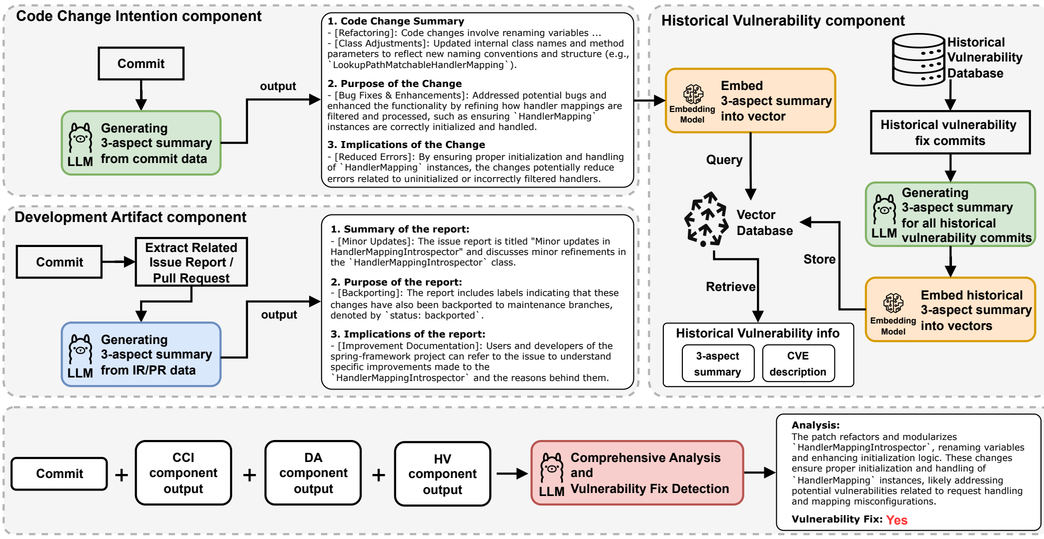

Fig. 4. The framework of LLM4VFD.

Challenge #2, the DA component leverages the information extracted from related development artifacts (*i.e.,* issue reports (IR) or pull requests (PR) in our case) to provide additional information to enrich the context for a commit. To address Challenge #3, the HV component builds a vector database and retrieves similar vulnerability fix commits from historical vulnerability data. The information from these components is then combined to enhance the analysis in the CAVFD component, where a comprehensive prompt template is employed to guide the LLM through all the relevant aspects of the commit. This structured prompt not only predicts whether the given commit is a vulnerability fix, but more importantly, it generates an in-depth explanation in analysis to assist security experts in understanding the rationale behind the prediction.

As highlighted in Figure 4, we provide a running example to illustrate the procedure of LLM4VFD. The outputs are taken from the commit from Challenge #1, which by leveraging our approach, can be correctly identified as a VF commit.

4.1 Code Change Intention (CCI)

The Code Change Intention component takes raw commit data (*i.e.,* the code diff and commit message) as input, and outputs a structured summary that abstracts the intention behind the commit. However, obtaining this information is not trivial, as the commit data does not explicitly provide this level of structured reasoning. To generate such information, we leverage and enhance the reasoning ability of LLMs using CoT techniques. Specifically, we design a structured prompt that breaks down the reasoning process into steps, mimicking how a security expert would analyze the commit. The prompt template, as shown in Figure 5, guides the LLM to analyze the commit code changes and distill relevant information into three key aspects (referred to as *3-aspect summary*): the summary, the purpose, and the implications as below.

(1) **Code Change Summary.** In the first step of the prompt, we instruct the LLM to focus on identifying the primary action of the code change. The goal here is to abstract the core modification and summarize different kinds of code changes.
(2) **Purpose of the Change.** The second step of the prompt instructs the LLM to reason through why the change was made. This step categorizes the commit into broader categories such as refactoring, feature enhancement, or fixing a vulnerability.

(3) **Implications of the Change.** In the final step, the prompt asks the LLM to consider the broader consequences of the change, including its potential impact. This step ensures that any security-related modifications are properly captured and that the potential risks or fixes introduced by the commit are fully understood.

In addition, to ease the extraction process for further components, we provide concrete instructions with a demonstration example to ensure LLM outputs the 3-aspect summary in the format we expect. Note that one commit, such as tangled commits, probably has multiple points of summary, purposes, and implications. We instruct LLM to generate multiple points (i.e., Key Point/Optional Key Points) for each aspect if it is applicable.

Prompt Template: Code Change Intention

System Prompt: You are a helpful software developer assistant specializing in software development life-cycle to help other developers understand the characteristics of software patches.
User Prompt: You are given the following software patch: ***{Commit}***
Think step by step and provide an analysis describing the following characteristics.
1. Code Change Summary
2. Purpose of the Change
3. Implications of the Change
Provide the analysis in bullet point format for each characteristic. Each bullet point should start with a key point and then briefly describe a main idea or fact from the text. Ensure each point is concise and captures the essence of the main idea it's summarizing. Here is an example of the desired format:
1. Code Change Summary
- [Key Point]: <description>
- [Optional Key Point]: <description>
2. Purpose of the Change
- [Key Point]: <description>
- [Optional Key Point]: <description>
3. Implications of the Change
- [Key Point]: <description>
- [Optional Key Point]: <description>

Fig. 5. The prompt template of Code Change Intention.

4.2 Development Artifact (DA)

In the DA component, we aim to integrate the information of external development artifacts such as issue reports (IRs) and pull requests (PRs) to enrich the analysis of a given commit. However, these artifacts can be lengthy and include irrelevant details, such as issue report templates and CI/CD notifications. To address this, and inspired by the CCI component in Section 4.1, we design a structured prompt that guides the LLMs through the process of analyzing and summarizing the related IRs and PRs to generate a 3-aspect summary which is similar to that for a commit. More specifically, the DA component generates a concise summary that abstracts the intention behind these artifacts by focusing on three core aspects: the summary, the purpose of the changes, and the potential implications. The prompt template is shown in Figure 6.

4.3 Historical Vulnerability (HV)

In this component, we aim to leverage information from historical vulnerability fix commits. More specifically, given a commit, HV aims to retrieve similar vulnerability fix commits to enrich the given commit. As discussed in Section 3, the vulnerability fix commit can be multi-purpose and contain code addressing other issues. Therefore, we decide to retrieve similar vulnerability fix commits based on their intention. To do this, we need to construct a database of historical vulnerability fixes. First, we collect historical vulnerability fix commits from existing vulnerability databases such

Prompt Template: IR/PR Summary

System Prompt: You are a helpful software developer assistant specializing in software development lifecycle to help other developers understand characteristics of software components such as patches, issue reports, pull requests, etc.
User Prompt: You are given the following Github issue report title and body information in JSON format which is related to a commit:***{Commit}***
Think step by step and provide an analysis describing the following characteristics.
1. Summary of the report
2. Purpose of the report
3. Implications of the report
Provide the analysis in bullet point format for each characteristic. Each bullet point should start with a key point and then briefly describe a main idea or fact from the text. Ensure each point is concise and captures the essence of the main idea it's summarizing. Include 1-3 key points. Here is an example of the desired format:
1. Summary of the report:
- [Key Point]: <description>
- [Optional Key Point]: <description>
2. Purpose of the report:
- [Key Point]: <description>
- [Optional Key Point]: <description>
3. Implications of the report:
- [Key Point]: <description>
- [Optional Key Point]: <description>

Fig. 6. The prompt template of Development Artifact.

as NVD [43]. We generate the 3-aspect summary for all the collected vulnerability fix commits using the CCI component. We then embed and vectorize the generated three-aspect summary for each vulnerability fix commit using a sentence embedding model [34], and store them in a vector database. Alongside the generated summaries, we also store their corresponding vulnerability metadata, including the CVE ID and CVE description.

When processing a new commit, we begin by using the CCI component to obtain a 3-aspect summary for the commit. We search for the nearest instance from historical vulnerability database, based on the similarity of their 3-aspect summary. With the nearest instance, we gather the 3-aspect summary and CVE description from its metadata as the output of the HV component.

4.4 Comprehensive Analysis and Vulnerability Fix Detection (CAVFD)

In the final step, we combine multiple sources of output collected from the three components (*i.e.,* CCI, DA, and HV), together with the commit's code change and commit message, into a comprehensive prompt for the LLM. This step ensures that the model has access to all relevant characteristics of the commit, not only in terms of the code itself, but also the surrounding context and historical vulnerability fix commits. The output is a prediction with a structured analysis and reasoning on how the decision is made.

The prompt template is shown in Figure 7. The prompt is designed to follow a multi-dimensional approach [7] to ensure a thorough, structured analysis for a given commit. We do this by guiding the LLM to analyze and integrate information from the components using the CoT and ICL techniques before making its decision. We first provide the raw commit data (code diff and commit message) as the patch content, following by the output from the three components. Next, we design the prompt to include a two-step task for the LLM: Comparison and Analysis. In the Comparison phase, we prompt the LLM to evaluate the current patch against the retrieved historical fixes, to avoid potential bias and ensure an evidence-based analysis. This is because we cannot ensure that the retrieved historical vulnerabilities are actually relevant to the current commit. In the Analysis step, we ask the LLM to synthesize information from the three components to determine whether the patch is a vulnerability fix, and to provide justification. Finally, we instruct the LLM to generate the response in JSON format including its detailed analysis and finally its decision. The analysis process is designed to output an analysis that can assist the user of LLM4VFD in the manual screening process to help them better understand the decision-making process of the LLM. We also conduct a user study in Section 6.3 to evaluate the usefulness of the resulting generated analyses.

Prompt Template: Comprehensive Analysis and Vulnerability Fix Detection

System Prompt: You are a helpful software developer assistant specializing in vulnerability detection to help other developers understand characteristics of software patches and discover potential vulnerabilities.
User Prompt: You are given the following details for analysis:
1. Patch Content: ***{Commit}***
2. Related Issue Report / Pull Request Summary: ***{DA component output}***
3. Three Aspect Analysis of the Patch: ***{CCI component output}***
4. Similar Historical Vulnerability Fix Information: ***{HV component output - CVE description}***
5. Three Aspect Analysis of the Historical Vulnerability Fix: ***{HV component output - 3-aspect summary}***
Task:
1. Comparison:
- Carefully compare the current patch with the historical vulnerability fix to avoid bias.
- Ensure that you consider the similarities and differences highlighted in the three aspect analyses.
2. Analysis:
- Use the information from the Related Issue Report / Pull Request Summary to understand the context and motivation behind the patch.
- Determine whether the current patch is intended to fix a vulnerability. You must provide evidence if you think its a vulnerability fix.
Your output should follow below syntax:
{"analysis": "<Detailed analysis of whether the patch is to fix a vulnerability>",
"vulnerability_fix": "<yes or no>"}

Fig. 7. The prompt template of Comprehensive Analysis and Vulnerability Fix Detection.

5 EXPERIMENTAL SETTINGS

In this section, we present research questions (RQs), datasets, evaluation metrics, our analysis approach for RQs, and implementation details.

5.1 Research Questions

We evaluate LLM4VFD in different aspects to answer the following research questions.

- *RQ1: How effective is LLM4VFD compared with SOTA techniques?*
- *RQ2: How effective is each component in LLM4VFD?*
- *RQ3: Can the analysis generated by LLM4VFD help security experts in identifying vulnerability fixes?*
- *RQ4: Bad Case Analysis: In which scenarios does LLM4VFD fail?*

5.2 Data Collection

5.2.1 Date Range Selection. LLMs are trained on extensive data, which results in a knowledge cutoff date reflecting the most recent information they possess. For our task, feeding LLMs with historical vulnerabilities predating their knowledge cutoff can lead to data leakage, as the model might already be aware of these vulnerabilities. As a mitigation, we restrict our analysis to vulnerabilities after 2023 to ensure that the data used is post-knowledge cutoff and reduces the risk of data leakage.

5.2.2 Vulnerability and Non-vulnerability Fix Commit Selection. We begin the data collection process by collecting historical CVE data from NVD [43]. NVD contains a vast amount of vulnerability data covering numerous open-source software (OSS) that cover a large spectrum of development activities and purposes. We only include CVEs that possess GitHub commit URLs in their references, which indicate a vulnerability fix. To obtain this info, we use GitHub's REST API endpoint to directly access repository and the language key in the response JSON [12]. To avoid the long tail effect from the diversity of vulnerabilities, we limit ourselves to vulnerabilities from 7 programming languages, namely Java, C, C++, Rust, JavaScript, Python, and Go.

In real-world open-source software (OSS) development, VF commits are exceptionally rare, comprising only a tiny fraction of total commits. The ratio of VF to normal commits can be as low as 1 in 1,000. For example, in the OSS project FFmpeg, we collected 114,210 total commits, of which only 124 were VF commits (0.1%). This extreme class imbalance makes our task significantly more challenging than typical binary classification tasks, which often assume a more balanced distribution (close to a 1:1 ratio). To address this imbalance and reduce the number of NVF commits evaluated, we followed the sampling strategy from the Big-Vul dataset [16] and selected a ratio of 1:16 between VF and NVF commits. We randomly sample NVF commits from the same OSS where we collected vulnerability fix commits.

Our final evaluation dataset BigVulFixes consists of 1,689 VF and 26,468 NVF commits, reflecting this sampled ratio. Additionally, to handle extreme outliers and ensure compatibility with the maximum token length of the language models we use, we excluded patches longer than the 99th percentile of patch token lengths (approximately 30,000 tokens).

5.3 IR/PR Data Collection

For all commits gathered from the previous steps, we collect related IR/PR URLs and their information using two approaches. The first approach parses the commit message to find any references to IR/PRs. GitHub uses an autolink feature via specific formatting syntax, allowing developers to reference IR/PR information directly in commit messages [12]. We design a regular expression to find such references to IR/PRs based on the defined autolink formatting syntax. The second approach accesses the "List pull requests associated with a commit" GitHub REST API endpoint. This endpoint provides the pull request the corresponding commit is included in. This method is particularly useful for cases where developers do not reference the related pull request in the commit message, causing the first approach to fail. Finally, we use the GitHub REST API to retrieve the information from each of the collected IR/PR URLs. We collected a total of 17,791 IR/PR data for the BigVulFixes dataset.

5.4 Historical Vulnerability Data Collection

In the previous section, we describe how we collect data after 2023 for evaluation, therefore, to avoid data leakage, the data range we select for the history vulnerability dataset will be all vulnerable data before 2023. Similar to the previous section, we start by collecting all history CVE information on NVD [43] before 2023 and collected 22,745 vulnerabilities from vulnerability advisory NVD. For each historical vulnerability, we collect its associate vulnerability fix commit and CVE description.

5.5 Implementation Detail

5.5.1 LLMs selection. LLM4VFD is a framework that can be implemented with any LLM and embedding models. In this study, we selected SOTA LLMs based on their performance on both code-related benchmarks (*e.g.,* HumanEval [8], MBPP EvalPlus [35]) and general benchmarks (*e.g.,* MMLU [24], IFEval [70]). We focused on three LLM families: Llama [15], Qwen [67], and Deepseek [77]. For the Llama family, we chose Llama3.1-70B and Llama3.1-8B. From the Qwen family, we selected Qwen2-70B and Qwen2-7B. For the Deepseek family, we included Deepseek-Coder-V2 (236B) and its smaller version, Deepseek-Coder-V2-Lite (16B). We use the "instruct" fine-tuning version of all LLMs.

We deployed Llama, Qwen and Deepseek-Coder-V2-Lite using vLLM [31] on Ascend 910 NPUs. For Deepseek-Coder-V2, we utilized the official API provided by DeepSeek. The total computational cost of the experiment was approximately 2.5 billion tokens. While more powerful LLMs like Llama3.1-405B or GPT-4o are available, they are either closed-source or too computationally intensive to deploy for our study. Therefore, we selected the models mentioned above.

5.5.2 Embedding Model and RAG implementation. For the embedding model used in our Retrieval-Augmented Generation (RAG) to embed the 3-aspect summary, we chose gte-Qwen2-7B-instruct [34]. This model was the state-of-the-art sentence embedding model on the Massive Text Embedding Benchmark [38] as of June 16, 2024. We use ChromaDB [11] to implement the vector database to use as part of HV and RAG. When querying the HV database, We filter the HV query results using a condition to retrieve historical vulnerabilities only with the same programming language of the current commit. We use the default Euclidean Distance function from ChromaDB to calculate and retrieve the most similar historical vulnerability.

5.6 Evaluation Metrics

We use precision, recall, F1-score and Matthews correlation coefficient (MCC) [37] as our evaluation metrics, which are widely used in previous studies [5, 33, 56, 68, 75]. Unlike previous research, we avoid using accuracy as an evaluation metric due to the highly imbalanced nature of vulnerability fix detection datasets, where the majority class can dominate and lead to misleading results [23].

5.7 RQ Approaches

5.7.1 RQ1: How effective is LLM4VFD compared with SOTA techniques? In RQ1, we compare LLM4VFD with existing SOTAs, including three PLM-based techniques and three selected LLMs with different sizes under the CoT 0-shot setting.

First, we select three SOTA PLM-based techniques, including VulFixMiner [72], CoLeFunda [71] and VulCurator [40]. We select VulFixMiner and CoLeFunda since they are the most commonly used baseline for vulnerability fix detection [41, 51]. VulCurator extends the approach proposed in VulFixMiner to include commit related development artifacts (*i.e.,* IR/PRs, commit message). Due to its design, CoLeFunda's implementation is limited to the Java language, therefore, we only compare CoLeFunda with LLM4VFD on Java data. We obtain these SOTA models as provided from the original authors, and use them according to guidance from the authors and or replication packages. Note that we do not reuse the VulFixMiner dataset which was used to evaluate both VulFixMiner and CoLeFunda since this dataset poses a threat of data leakage since it contains data before 2023, which predates the cutoff dates of all LLMs included in our study. Therefore, we evaluate all models using our newly collected dataset BigVulFixes.

Secondly, we also compare the LLMs under their vanilla setting without LLM4VFD, defined as the 0-shot CoT setting. For a fair comparison, we maintain the structure of the vulnerability fix detection prompt template as shown in Figure 7, but omit all information from the Code Change Intention (CCI), Development Artifact (DA), and History Vulnerability (HV) components. In other words, the LLM is only given the patch content and is tasked with determining whether the commit is a VF, without any additional context from LLM4VFD. Additionally, the task instruction is simplified to:

"Determine whether the current patch is intended to fix a vulnerability. You must provide evidence if you think it's a vulnerability fix".

5.7.2 RQ2: How effective is each component in LLM4VFD? In our framework, we integrate three key components: Code Change Intention (CCI), Development Artifacts (DA), and Historical Vulnerability (HV). To understand the contribution of each component, we conduct an ablation study to evaluate their individual impact on the overall performance of LLM4VFD. For this study, we selected two models, Qwen2-72B and Qwen-7B, from the Qwen family. These models were chosen because both the larger and smaller versions demonstrated strong performance in our initial experiments (see section 6.1 for details). In the ablation study, we systematically removed each component one at a time and compared the performance of the models with and without the removed component.

5.7.3 RQ3: Can the analysis generated by LLM4VFD help security experts in identifying vulnerability fixes? Having a model detect and label a commit as a VF is generally not the end of the story. Security experts will often perform screening on the instance to confirm the prediction was correct. This is an essential step before beginning remediation, such as applying the patch into downstream software dependencies. Therefore, in RQ3 we conduct a user study to investigate whether analysis generated by LLM4VFD can help developers identify vulnerability fixes more effectively. We elaborate on the methodology of our user study below.

Participants: We invited 10 security experts from industry and academia with 3–5 years of software security experience for the user study from industry. We also ensure that these security experts experience of screening vulnerability fixes before.

Task: We randomly selected 40 VF and their analysis generated by LLM4VFD based on Qwen2-72B.

Procedure: Each participant is tasked to answer a sequence of yes or no questions with the option of adding additional comments given a VF and the generated analysis from LLM4VFD.

We present the analysis result generated by LLM4VFD, and ask reviewers to answer the following questions:

(1) Does the analysis help understand the intent behind the code changes?
(2) Does the analysis accurately characterize the vulnerability?
(3) Does the analysis provide a better understanding of the root cause of the vulnerability?
(4) Does the analysis provided by the large model help improve the efficiency of identifying vulnerabilities fix?
(5) Are you satisfied with the quality of the generated content (*e.g.,* it contains redundant information, inaccurate information, not thorough enough, hallucination, historical vulnerabilities mentioned are irrelevant to the vulnerability, etc.)

By collecting the answers to these questions, we aim to determine if the analysis information provided by LLM4VFD can help participants effectively understand the commit and verify vulnerabilities fixes.

5.7.4 RQ4: Bad Case Analysis: In which scenarios does LLM4VFD fail? To better understand the limitations of LLM4VFD, we conduct a manual analysis on failed predictions, focusing on two types of misclassifications: (1) False Positives (FP): Cases where LLM4VFD incorrectly classifies a commit as a VF. This helps us examine scenarios where LLM4VFD misinterprets the intent or context of the commit. And (2) False Negatives (FN): Cases where LLM4VFD fails to identify a commit as a VF. This analysis highlights situations where LLM4VFD overlooks important indicators of a VF.

The first three authors each conducted a manual inspection on 20 FP and 20 FN for each of Qwen-72B and Qwen-7B following RQ2. A total of 240 cases has been reviewed. We aim to identify the specific reasons for the misclassifications. By pinpointing the root causes of the errors, we seek to understand the current limitations of LLM4VFD and identify room for future improvement.

Table 1. The performance of vulnerability fix detection approaches.

Foundation Model	Parameter Size	Approach	Precision	Recall	F1-score	MCC
CodeBERT	125M	VulFixMiner	0.17	0.26	0.20	0.14
	125M	CoLeFunDa *	0.50	0.06	0.11	0.15
	125M	VulCurator	**0.77**	0.13	0.22	0.30
Deepseek-Coder-V2	236B	Vanilla	0.33	**0.84**	0.47	0.48
		LLM4VFD	0.40	0.78	0.53	**0.53**
	16B	Vanilla	0.23	0.44	0.30	0.26
		LLM4VFD	0.28	0.65	0.39	0.37
Llama3.1	70B	Vanilla	0.41	0.53	0.47	0.43
		LLM4VFD	0.49	0.61	**0.54**	0.52
	8B	Vanilla	0.10	0.86	0.18	0.18
		LLM4VFD	0.30	0.50	0.37	0.34
Qwen2	72B	Vanilla	0.32	0.72	0.44	0.43
		LLM4VFD	0.38	0.77	0.51	0.50
	7B	Vanilla	0.24	0.48	0.32	0.29
		LLM4VFD	0.52	0.48	0.50	0.47

* Due to the tool limitation, CoLeFunDa is evaluated on only Java vulnerabilities.

6 RESULTS

6.1 RQ1 - Effectiveness

6.1.1 PLM-based Approaches vs. LLM4VFD. **LLM4VFD outperforms PLM-based approach consistently in terms of MCC, F1-score, recall. For instance, LLM4VFD outperforms the best-performed PLM-based approach VulCurator by 68.1% - 145.4% across LLMs in terms of F1-score.** As shown in Table 1, LLM4VFD significantly outperforms all three PLM-based approaches, VulFixMiner, CoLeFunda, and VulCurator, in terms of F1-score, MCC, and Recall. LLM4VFD with various LLMs achieves an F1-score ranging from 0.37 to 0.54, which is significantly higher than PLM-based approaches with an F1-score of 0.14 to 0.22. For instance, the best performed LLM4VFD (with Llama3.1-70B) achieves an F1-score of 0.54, exhibiting a 145.5% improvement in F1-score compared to the best PLM-based approach VulCurator. Although VulCurator and ColeFunDa achieve good precision 0.50 and 0.77 respectively, they suffer from very low Recall of 0.06 and 0.13 and miss many true vulnerability fixes, which hinders their practical application.

6.1.2 Vanilla LLMs (0-shot CoT) vs. LLM4VFD. **LLM4VFD outperforms all vanilla LLMs in terms of F1-score and MCC.** Compared to the vanilla LLM, we observe that LLM4VFD improves performance across all evaluation metrics, except Recall on two LLMs (*i.e.,* Deepseek-Coder-V2 and Llama3.1-8B-Instruct). In terms of F1-score, for each LLM LLM4VFD achieves a range of 12.7% to 105.6% improvements over its vanilla setting. We observe a similar pattern of improvement for MCC. In particular, LLM4VFD with Deepseekcoder-V2 achieves a Precision of 0.40 and a Recall of 0.78, resulting in an F1-score of 0.53, representing a 13% improvement in Precision and a 12.8% increase in F1-score compared to its vanilla setting. Similarly, Llama3.1-70B improves its Precision from 0.41 to 0.49 (a 19.5% increase) and Recall from 0.53 to 0.61 (a 15.1% increase), resulting in an F1-score of 0.54 (a 14.9% increase).

Larger models typically outperform smaller models, while smaller models benefit more from our framework. For instance, F1-score for smaller models achieve an improvement of 64.0% on average after applying LLM4VFD compared with the vanilla setting, while large models achieve an average improvement of 14.4%. When comparing smaller models with their larger counterparts within the same family, we observed that the larger models consistently

Table 2. The ablation results. The cells with the lowest performance are marked in bold, indicating the largest contribution from that component.

Ablation Setting	Qwen2-72B				Qwen2-7B			
	Precision	**Recall**	**F1-score**	**MCC**	**Precision**	**Recall**	**F1-score**	**MCC**
LLM4VFD	0.38	0.77	0.51	0.50	0.52	0.48	0.50	0.47
w/o CCI component	**0.33**	0.77	**0.46**	**0.45**	0.40	0.54	**0.46**	**0.42**
w/o DA component	0.36	0.75	0.48	0.48	0.49	**0.47**	0.48	0.45
w/o HV component	0.36	**0.74**	0.49	0.48	**0.39**	0.62	0.48	0.45

outperform the smaller ones, both with and without the LLM4VFD framework. This is likely because vulnerability fix detection is a complex task that benefits from the extra parameters providing more code understanding capabilities. In addition, the explanations for LLMs in the vanilla setting show significant variation, with F1-scores ranging widely from 0.18 to 0.50, especially for smaller models. For example, within the Llama family, the vanilla Llama3.1-70B-Instruct model achieves an F1-score of 0.47, whereas the smaller Llama3.1-8B-Instruct model only achieves an F1-score of 0.18. However, after applying our LLM4VFD framework, we find that the improvement in performance is more pronounced for smaller models than for larger ones. On average, the F1-scores see a relative improvement of 64.0%, whereas larger models saw a more modest average increase of 14.4%. These results indicate that LLM4VFD significantly enhances the performance of smaller models, making them more competitive with larger models and narrowing the performance gap.

LLM4VFD outperforms PLM-based approach consistently in terms of MCC, F1-score, and recall. For instance, LLM4VFD outperforms the best-performed PLM-based approach VulCurator by 68.1%–145.4% across LLMs in terms of F1-score. Our framework demonstrates performance gains ranging from 12.7% to 105.6% over its vanilla variant, with smaller models generally benefiting more compared to their larger counterparts.

6.2 RQ2 - Ablation analysis.

Overall, all three components in LLM4VFD make positive contributions to the overall performance. Among them, Code Change Intention have a larger impact than Development Artifact and Historical Vulnerability. Table 2 shows the results of our ablation analysis. The removal of CCI component leads to a significant decrease in performance, particularly in terms of Precision and F1-score. For Qwen2-72B, the Precision drops from 0.38 to 0.33, representing a 13.1% reduction. In Qwen2-7B, the decline is similar, with the Precision decreasing from 0.52 to 0.40, an 15.4% reduction. These results emphasize the important role of CCI in improving the model's ability to capture vulnerability fix commits more accurately by reducing false positives, and the improvement is especially notable in smaller models, where CCI helps balance Precision and Recall effectively. DA component has a more similar effect on Qwen2-7B and Qwen2-72B. In Qwen2-72B, the F1-score improves significantly from 0.48 to 0.51, marking an 6.3% increase, indicating that the additional contextual information provided by DA substantially boosts both precision and recall. Removing the HV component or DA components results in slight reductions in F1-score and MCC for both models. However, we find that Qwen2-7B sees a more significant reduction in precision without the HV components, dropping from 0.52 to 0.39, a 25% decrease.

The ablation study highlights the importance of each component in LLM4VFD, particularly CCI, which demonstrates significant improvements in precision and F1-score, especially for smaller models like Qwen2-7B. HV further enhances precision by reducing false positives, albeit with a more moderate effect. Collectively, these components provide a comprehensive framework that significantly enhances the accuracy and reliability of vulnerability fix detection, particularly by augmenting smaller models with crucial contextual and historical information.

All components in LLM4VFD make positive contributions to the performance. The impact from Code Change Intention is larger than Development Artifact and Historical Vulnerability.

6.3 RQ3 - User study

The analysis generated by LLM4VFD helps security experts to understand the intent of code changes and the vulnerabilities, which improves the efficiency of identifying vulnerability fixes. Our result shows that in 95.0% (38 out of 40) of the cases, LLM4VFD 's analysis can help with understanding the intent of commits (Question 1). A case like CVE-2024-29199 [48] has 47 changed files with 517 lines of addition and 226 lines of deletion. It is too long for participants to understand, however LLM4VFD's analysis helped participants to understand the change. In addition, participants think that LLM4VFD accurately describes the characteristics of the vulnerabilities in 90% of the commits (Question 2) and the root causes in 75.0% (30 out of 40) of the cases (Question 3). With LLM4VFD, participants can identify VF commit more efficiently. This is evidenced by the fact that in 80.0% (32 out of 40) of the commits, LLM4VFD improved their efficiency in identifying vulnerability fixes (Question 4). We investigate the 8 cases where participants did not think the analysis generated by LLM4VFD helped them to identify vulnerability fixes more efficiently, 7 of them are due to the vulnerability fixes are easy to identify even without the help of LLM4VFD. In one case of CVE-2024-28103 [47], the participant did not understand the vulnerability even with the help of LLM4VFD.

When investigating the feedback we collected related to the quality of the analysis generated by LLM4VFD (Question 5), we find that in 75% of the cases, participants are satisfied with the overall quality of the analysis. There were some cases (*e.g.,* CVE-2023-48014 [45] and CVE-2023-37061 [44]) where participants noted that the analysis contained redundant information, or lacked depth analysis for more complex vulnerabilities (*e.g.,* CVE-2023-48657 [46]). This suggests a need for further refinement in handling edge cases and ensuring that the historical context provided is directly relevant and concise. Nevertheless, ours is the first work on this direction and more future research is encouraged.

Overall, the user study shows that the analysis generated by LLM4VFD helps security experts to understand the intent of code changes and the vulnerabilities, which improves the efficiency of identifying vulnerability fixes.

6.4 RQ4 - Failure analysis

Table 3 summarizes the results of our failure analysis. To better understand the misclassification issues encountered by LLM4VFD, it is crucial to clarify the relationship between security fixes and vulnerability fixes. A *vulnerability fix* is a subset of *security fixes*. Security fixes encompass any code changes that enhance the overall security of the software, such as improvements to authentication mechanisms, encryption implementations, or adherence to security best practices, whereas vulnerability fixes aim to address security flaw, glitch, or weakness found in software code that could be exploited by an attacker [43]. Therefore, in our dataset, a security fix commit that does not aim to fix a vulnerability is considered as non-vulnerability fix (i.e., *non-vulnerability security fix*), although they are related to security fix.

The most mis-classification made by LLM4VFD due to its struggling to distinguish between vulnerability fix and a non-vulnerability security fix. For instance, in false positive (FP) cases, 75.0% of FP in Qwen2-72B and 60.0% in Qwen2-7B occured due to non-vulnerability security fixes are misclassified as vulnerability fixes. Vise versa, it led to 46.7% and 55.0% of the FN cases

Table 3. Bad Case Analysis on Qwen2 Models

Type	Reason	72B	7B
FP	Potential unreported vulnerability fix	1	7
	Non-vulnerability security fix misclassified as vulnerability fix	45	36
	Non-functional Change	7	8
	Fail to realize the change is not related to security	4	5
	Mislead by retrieved similar vulnerability	2	3
	Others	1	1
FN	Vulnerability fix misclassified as non-vulnerability security fix	28	33
	Unable to identify security related code change	21	18
	Mislead by retrieved similar vulnerability	5	6
	Unable to pinpoint vulnerability related code change from long context	1	0
	Others	5	3

where vulnerability fixes are misclassified as non-vulnerability fixes in Qwen2-72B and Qwen2-7B. The result indicates that even if LLM4VFD can identify security-related commits, it sometimes fails to interpret their severity. Specifically, in the analysis of an FN case, the LLM explains with: "Although changes are related to security, I am uncertain if the changes fix a vulnerability". One possible explanation is that the LLM is unable to differentiable between security related change and vulnerability fix when generating the intention for commits. Although, LLM4VFD struggles to distinguish between vulnerability fix and non-vulnerability security fix. Both vulnerability fix and non-vulnerability security fix are security fixes, and identifying them is beneficial.

Another frequent failure in FN cases is the inability to identify security related code change (32.5%). Another notable failure case is related to its use of historical vulnerability data through the HV component. The retrieval of irrelevant history vulnerability causes misclassification in both FP and FN cases. Specifically, in 4.2% of the FP cases, and 9.2% of the FN cases, the HV component retrieved vulnerabilities that seemed relevant but did not match the functional or contextual details of the current commit, and such information caused confusion to LLM, leading LLM4VFD to make incorrect assumptions on the commits. While our HV component improves the overall performance (as shown in our ablation study in Section 6.2), it is not perfect and can sometimes retrieve irrelevant historical vulnerabilities. Future studies can focus on this aspect by developing better retrieval techniques and improving the underlying vulnerability database.

We also find in a few FP cases where the commit is potentially fixing an unreported vulnerability. For example, in one of the checked cases, the commit is a merge of changes from a GHSA [20] pull request. GHSA is an independent security advisor that maintains its own vulnerability dataset and may have different assessments compared to NVD.

7 DISCUSSION

7.1 Potential Future Direction

Our research is the first study to explore vulnerability fix detection using LLMs. However, insights from our user study and failure case analysis indicate several areas for future exploration. First, incorporating more relevant project-specific information, such as security policies and historical commit patterns, could help LLMs better distinguish vulnerability fixes among security-related commits. Second, the prompt templates guiding the LLM could be refined to more effectively leverage context and produce more insightful analysis results for security experts. Finally, the current RAG design is basic, and incorporating advanced techniques — such as dynamic retrieval strategies or reranking mechanisms — could improve the precision and relevance of retrieved historical vulnerabilities.

7.2 Threats to Validity

7.2.1 Internal Validity. Due to the extremely imbalanced nature of vulnerability fixes and non-vulnerability fixes, when our approach is applied to monitor real-world projects there may be a large amount of false negatives. Based on feedback from our industry partners, the false positive rate was considered acceptable, as it requires only a small amount of additional human effort to review the results. Previous studies suggest that LLM settings, such as temperature, have an impact on outputs [53, 57]. In this study, we use the default settings for the studied LLMs for all RQs.

7.2.2 External Validity. Threats to external validity relate to the generalizability of our approach. Since LLM4VFD is a framework, and its implementation relies on LLMs. The effectiveness of LLM4VFD also depends on the performance and capabilities of the LLMs. To mitigate this threat, we evaluated LLM4VFD using several well-known and state-of-the-art LLMs, including both large models (70B–236B parameters) and smaller models (7B–16B parameters), and our results show that with all models, LLM4VFD outperforms SOTA baselines. Future research is encouraged to investigate the performance with more LLMs using our framework.

8 CONCLUSION

In this paper, we propose LLM4VFD, a novel framework that leverages Large Language Models (LLMs) enhanced with Chain-of-Thought reasoning and In-Context Learning to improve the accuracy of vulnerability fix detection by integrating information from multiple sources (e.g., related development artifacts and historical vulnerabilities). More importantly, on top of the prediction, LLM4VFD also provides a detailed explanation in analysis to help security experts understand the rationale behind the decision. Experimental results demonstrate that LLM4VFD outperforms PLM-based approach consistently in terms of MCC, F1-score, and recall. For instance, LLM4VFD outperforms the best-performed PLM-based approach VulCurator by 68.1% - 145.4% in terms of F1-score when using different LLMs as the base model. Furthermore, we conducted a user study involving security experts to assess the effectiveness of the analysis generated by LLM4VFD in aiding vulnerability fix identification. The feedback from the participants demonstrates that the analysis provided by LLM4VFD improves the efficiency of identifying vulnerability fixes.

DATA AVAILABILITY

We make all the datasets and code used in this study openly available in our replication package [1].

REFERENCES

[1] 2024. *LLM4VFD replication pacakge.* Zenodo. https://doi.org/10.5281/zenodo.13776994

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[3] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code. *arXiv preprint arXiv:2206.01335* (2022).

[4] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. 2015. Helping developers help themselves: Automatic decomposition of code review changesets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*.

[5] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering* (2021).

[6] Aaron Chan, Anant Kharkar, Roshanak Zilouchian Moghaddam, Yevhen Mohylevskyy, Alec Helyar, Eslam Kamal, Mohamed Elkamhawy, and Neel Sundaresan. 2023. Transformer-based vulnerability detection in code at EditTime: Zero-shot, few-shot, or fine-tuning? *arXiv preprint arXiv:2306.01754* (2023).
[7] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. 2023. Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review. *arXiv preprint arXiv:2310.14735* (2023).
[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[9] Yang Chen, Andrew E Santosa, Ang Ming Yi, Abhishek Sharma, Asankhaya Sharma, and David Lo. 2020. A machine learning approach for vulnerability curation. In *Proceedings of the 17th International Conference on Mining Software Repositories*.
[10] Yiran Cheng, Lwin Khin Shar, Ting Zhang, Shouguo Yang, Chaopeng Dong, David Lo, Shichao Lv, Zhiqiang Shi, and Limin Sun. 2024. LLM-Enhanced Static Analysis for Precise Identification of Vulnerable OSS Versions. *arXiv preprint arXiv:2408.07321* (2024).
[11] Chroma. 2024. Chroma is the open-source AI application database. Batteries included. https://www.trychroma.com/ accessed 2024-09-12.
[12] GitHub Docs. 2022. GitHub REST API documentation. https://docs.github.com/en/rest?apiVersion=2022-11-28 accessed 2024-08-19.
[13] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234* (2022).
[14] Xueying Du, Geng Zheng, Kaixin Wang, Jiayi Feng, Wentai Deng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2024. Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG. *arXiv preprint arXiv:2406.11147* (2024).
[15] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).
[16] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*.
[17] Lishui Fan, Jiakun Liu, Zhongxin Liu, David Lo, Xin Xia, and Shanping Li. 2024. Exploring the Capabilities of LLMs for Code Change Related Tasks. *arXiv preprint arXiv:2407.02824* (2024).
[18] Richard Fang, Rohan Bindu, Akul Gupta, Qiusi Zhan, and Daniel Kang. 2024. Teams of LLM Agents can Exploit Zero-Day Vulnerabilities. *arXiv preprint arXiv:2406.01637* (2024).
[19] Henry Gilbert, Michael Sandborn, Douglas C Schmidt, Jesse Spencer-Smith, and Jules White. 2023. Semantic compression with large language models. In *2023 Tenth International Conference on Social Networks Analysis, Management and Security (SNAMS)*.
[20] GitHub. 2024. GitHub Advisory Database. https://github.com/advisories accessed 2024-08-19.
[21] gpac. 2023. fixed #2475 · gpac/gpac@c88df2e · GitHub. https://github.com/gpac/gpac/commit/c88df2e202efad214c25b4e586f243b2038779ba accessed 2024-08-19.
[22] gpac. 2023. OOB Read segfault · Issue #2475 · gpac/gpac · GitHub. https://github.com/gpac/gpac/issues/2475 accessed 2024-08-19.
[23] Haibo He and Edwardo A Garcia. 2009. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering* (2009).
[24] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300* (2020).
[25] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*.
[26] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620* (2023).
[27] Nafis Tanveer Islam, Joseph Khoury, Andrew Seong, Gonzalo De La Torre Parra, Elias Bou-Harb, and Peyman Najafirad. 2024. LLM-Powered Code Vulnerability Repair with Reinforcement Learning and Semantic Reward. *arXiv preprint arXiv:2401.03374* (2024).
[28] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. Selfevolve: A code evolution framework via large language models. *arXiv preprint arXiv:2306.02907* (2023).
[29] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* (2022).

[30] Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d'Amorim. 2024. A Case Study of LLM for Automated Vulnerability Repair: Assessing Impact of Reasoning and Patch Validation Feedback. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*.
[31] 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. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.
[32] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. 2023. Comparing code explanations created by students and large language models. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*.
[33] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
[34] Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281* (2023).
[35] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* (2024).
[36] Shigang Liu, Guanjun Lin, Lizhen Qu, Jun Zhang, Olivier De Vel, Paul Montague, and Yang Xiang. 2020. CD-VulD: Cross-domain vulnerability discovery based on deep domain adaptation. *IEEE Transactions on Dependable and Secure Computing* (2020).
[37] Brian W Matthews. 1975. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure* (1975).
[38] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. 2022. MTEB: Massive text embedding benchmark. *arXiv preprint arXiv:2210.07316* (2022).
[39] Son Nguyen, Thanh Trong Vu, and Hieu Dinh Vo. 2023. VFFINDER: A Graph-based Approach for Automated Silent Vulnerability-Fix Identification. In *2023 15th International Conference on Knowledge and Systems Engineering (KSE)*.
[40] 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*.
[41] Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Chengran Yang, Zhipeng Zhao, Bowen Xu, Jiayuan Zhou, Xin Xia, Ahmed E Hassan, et al. 2023. Multi-granularity detector for vulnerability fixes. *IEEE Transactions on Software Engineering* (2023).
[42] Giang Nguyen-Truong, Hong Jin Kang, David Lo, Abhishek Sharma, Andrew E Santosa, Asankhaya Sharma, and Ming Yi Ang. 2022. Hermes: Using commit-issue linking to detect vulnerability-fixing commits. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*.
[43] NIST. 2024. NVD - Home. https://nvd.nist.gov/ accessed 2024-09-10.
[44] NVD. 2023. NVD - CVE-2023-37061. https://nvd.nist.gov/vuln/detail/CVE-2024-37061 accessed 2024-08-25.
[45] NVD. 2023. NVD - CVE-2023-48014. https://nvd.nist.gov/vuln/detail/CVE-2024-48014 accessed 2024-08-25.
[46] NVD. 2023. NVD - CVE-2023-48657. https://nvd.nist.gov/vuln/detail/CVE-2024-48657 accessed 2024-08-25.
[47] NVD. 2024. NVD - CVE-2024-28103. https://nvd.nist.gov/vuln/detail/CVE-2024-28103 accessed 2024-08-25.
[48] NVD. 2024. NVD - CVE-2024-29199. https://nvd.nist.gov/vuln/detail/CVE-2024-29199 accessed 2024-08-25.
[49] OpenSSF. 2024. Guide to coordinated vulnerability disclosure for open source software projects. https://github.com/ossf/oss-vulnerability-guide accessed 2024-07-26.
[50] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.
[51] Shengyi Pan, Lingfeng Bao, Xin Xia, David Lo, and Shanping Li. 2023. Fine-grained commit-level vulnerability type prediction by CWE tree structure. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.
[52] Shengyi Pan, Jiayuan Zhou, Filipe Roseiro Cogo, Xin Xia, Lingfeng Bao, Xing Hu, Shanping Li, and Ahmed E Hassan. 2022. Automated unearthing of dangerous issue reports. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
[53] Max Peeperkorn, Tom Kouwenhoven, Dan Brown, and Anna Jordanous. 2024. Is temperature the creativity parameter of large language models? *arXiv preprint arXiv:2405.00492* (2024).
[54] protobufjs. 2022. fix: do not let setProperty change the prototype (#1731) · protobufjs/protobuf.js@3357ef7 · GitHub. https://github.com/protobufjs/protobuf.js/commit/3357ef753871b394b825d15429ceb27b26e24d63 accessed 2024-08-19.
[55] protobufjs. 2023. fix: do not let setProperty change the prototype (#1899) · protobufjs/protobuf.js@e66379f · GitHub. https://github.com/protobufjs/protobuf.js/commit/e66379f451b0393c27d87b37fa7d271619e16b0d accessed 2024-08-19.

[56] Md Mahbubur Rahman, Ira Ceka, Chengzhi Mao, Saikat Chakraborty, Baishakhi Ray, and Wei Le. 2024. Towards causal deep learning for vulnerability detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*.

[57] Matthew Renze and Erhan Guven. 2024. The effect of sampling temperature on problem solving in large language models. *arXiv preprint arXiv:2402.05201* (2024).

[58] Antonino Sabetta and Michele Bezzi. 2018. A practical approach to the automatic classification of security-relevant commits. In *2018 IEEE International conference on software maintenance and evolution (ICSME)*.

[59] Da Shen, Xinyun Chen, Chenguang Wang, Koushik Sen, and Dawn Song. 2022. Benchmarking Language Models for Code Syntax Understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2022*.

[60] spring projects. 2023. Polishing and minor refactoring in HandlerMappingIntrospector · spring-projects/spring-framework@202fa5c · GitHub. https://github.com/spring-projects/spring-framework/commit/202fa5cdb3a3d0cfe6967e85fa167d978244f28a accessed 2024-08-19.

[61] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-based language models for software vulnerability detection. In *Proceedings of the 38th Annual Computer Security Applications Conference*.

[62] The New York Times. 2019. Equifax to Pay at Least $650 Million in Largest-Ever Data Breach Settlement. https://www.nytimes.com/2019/07/22/business/equifax-settlement.html accessed 2024-08-19.

[63] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*.

[64] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* (2022).

[65] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[66] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. Spain: security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*.

[67] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671* (2024).

[68] Xu Yang, Shaowei Wang, Yi Li, and Shaohua Wang. 2023. Does data sampling improve deep learning-based vulnerability detection? Yeas! and Nays!. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*.

[69] JD Zamfirescu-Pereira, Richmond Y Wong, Bjoern Hartmann, and Qian Yang. 2023. Why Johnny can't prompt: how non-AI experts try (and fail) to design LLM prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–21.

[70] Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. 2023. Instruction-following evaluation for large language models. *arXiv preprint arXiv:2311.07911* (2023).

[71] Jiayuan Zhou, Michael Pacheco, Jinfu Chen, Xing Hu, Xin Xia, David Lo, and Ahmed E Hassan. 2023. Colefunda: Explainable silent vulnerability fix identification. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

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

[73] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2024. Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead. *arXiv preprint arXiv:2404.02525* (2024).

[74] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. 2024. Out of Sight, Out of Mind: Better Automatic Vulnerability Repair by Broadening Input Ranges and Sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*.

[75] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* (2019).

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

[77] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint*

arXiv:2406.11931 (2024).