Towards Higher Quality Software Vulnerability Data Using LLM-based Patch Filtering

Charlie Dil[a], Hui Chen[b], Kostadin Damevski[a,*]

[a] *Virginia Commonwealth University, Richmond, Virginia, USA*
[b] *CUNY Brooklyn College, Brooklyn, New York, USA*

Abstract

High-quality vulnerability patch data is essential for understanding vulnerabilities in software systems. Accurate patch data sheds light on the nature of vulnerabilities, their origins, and effective remediation strategies. However, current data collection efforts prioritize rapid release over quality, leading to patches that are incomplete or contain extraneous changes. In addition to supporting vulnerability analysis, high-quality patch data improves automatic vulnerability prediction models, which require reliable inputs to predict issues in new or existing code.

In this paper, we explore using large language models (LLMs) to filter vulnerability data by identifying and removing low-quality instances. Trained on large textual corpora including source code, LLMs offer new opportunities to improve data accuracy. Our goal is to leverage LLMs for reasoning-based assessments of whether a code hunk fixes a described vulnerability. We evaluate several prompting strategies and find that Generated Knowledge Prompting, where the model first explains a hunk's effect, then assesses whether it fixes the bug, is most effective across three LLMs. Applying this filtering to the BigVul dataset, we show a 7–9% improvement in prediction precision for three popular vulnerability prediction models. Recall declines slightly, 2–8%, across models, likely reflecting the impact of reduced dataset size.

Keywords: Vulnerability Patch Quality, Automatic Vulnerability Prediction, Large Language Models

*Corresponding author

Email addresses: `ndil@vcu.edu` (Charlie Dil), `hui.chen@brooklyn.cuny.edu` (Hui Chen), `kdamevski@vcu.edu` (Kostadin Damevski)

1. Introduction

A wide range of government and industry initiatives, such as the National Vulnerability Database (NVD) [1], Snyk Vulnerability Database [2], and GitHub Security Advisories [3], aim to collect and disseminate information about vulnerabilities. However, these data sources primarily focus on rapidly sharing details about newly found vulnerabilities, which differs from the needs of researchers seeking large datasets for building machine learning models or gathering empirical evidence. Despite significant manual efforts by research groups to curate high-quality historic datasets from these vulnerability databases, challenges remain. These datasets are either limited in size or suffer from significant data quality issues, highlighting an urgent need for automated collection high-quality, large-scale software vulnerability data.

Large Language Models (LLMs) have become pivotal in harnessing artificial intelligence for a variety of purposes. Through unsupervised training on vast text corpora, LLMs exhibit emergent reasoning capabilities [4, 5, 6]. These models are interacted with via textual prompts that describe the desired task in natural language. To leverage LLMs for sophisticated reasoning tasks, researchers often employ intricate prompting strategies, which guide the LLM through a sequence of incremental reasoning steps. A number of prompting strategies have been proposed, including few-shot prompting, chain-of-thought prompting, and generated knowledge prompting, a recent technique that enhances LLM reasoning on downstream tasks by first prompting language models to generate relevant background knowledge or context, which is then integrated into the task [7].

In this paper, we investigate the following Research Questions (RQs):

RQ1: *Is advanced prompting of LLMs an effective approach for filtering software patch data for improving data quality?*

We employed LLMs to identify extraneous hunks, contiguous blocks of modified lines in a file, that do not contribute to fixing the bug. These extraneous hunks can result from incorrectly linked patches or unrelated changes that are tangled with the fix. This RQ is to test the LLMs' ability to filter out these low quality patches. We evaluated the performance of LLMs using various prompting techniques to examine patch changesets (or diffs). We found that generated knowledge prompting with GPT-4 is the most effective approach for patch filtering.

RQ2: *Does filtered software vulnerability patch data improve the performance of automated software vulnerability prediction?*

We hypothesize that the filtered software vulnerability via our approach can improve downstream applications of the data, such as software vulnerability prediction. Thus, for this RQ, we gauge the downstream impact of LLM-filtered vulnerability patch data. We compared the performance of popular software vulnerability prediction models trained on an unfiltered dataset to that of models trained on datasets filtered using an LLM. We observed consistent improvements in certain predictive performance metrics, namely, precision and AUC-PR, despite the overall smaller size of the filtered datasets. However, recall was generally lower in the filtered dataset, which is expected since filtering likely removes some relevant instances along with the noise.

In summary, the contributions of this paper are:

- a novel LLM-based vulnerability data filtering approach based on generative knowledge prompting;
- an understanding of the capability of combinations of LLMs and prompting strategies for vulnerability patch assessment and filtering through extensive evaluation;
- enhanced performance of vulnerability prediction models based on the filtered data;
- insights on the trade-off between dataset size and increased recall

While this work is motivated by challenges in vulnerability dataset quality, we use a manually-annotated bug-fix dataset for RQ1 to validate the effectiveness of LLM-based filtering, due to the absence of reliable hunk-level vulnerability annotations. We then apply the best-performing method to vulnerability data in RQ2 to assess its impact on real-world vulnerability prediction. A replication package containing our scripts, prompts, and relevant data is available at: `https://github.com/charliedil/LLM_based_patch_filtering`

2. Motivating Example

Software repositories are updated via changesets. Each changeset consists of a set of hunks, i.e., contiguous blocks of lines that have undergone addition, removal, or modification within a file [8]. This granular unit of change encapsulates specific modifications, serving as a coherent unit within the overall changeset. Typically, a hunk is computed by employing a diff algorithm, such as the Myers algorithm [9], which systematically compares differences between two file versions. This algorithm isolates sections with differing lines, grouping these differences into hunks and including contextual lines surrounding the changes to enhance readability and ensure accurate patching or merging processes.

In this paper, we are interested in determining whether hunks in a changeset are related to patching a security vulnerability or not. We operate at the hunk level since hunks are more easily comprehensible to developers as they group changes in adjacent lines and provide context, unlike individual lines which can often be too granular.

In Figure 1, we present several hunks from a single security patch. This patch is linked in the National Vulnerability Database (NVD), a comprehensive resource on software vulnerabilities that sometimes includes URLs to patches, which often serves as a key resource for curating security vulnerability data for training Software Vulnerability Prediction models. The CVE identifier that the NVD associates with this patch is CVE-2022-23472. For brevity, we include only a subset of hunks from this changeset. The patch updates the `Passeo` package to use the `Secrets` package instead of the `Random` package for enhanced security. The NVD describes this vulnerability as follows indicating the source of the security vulnerability, i.e., the dependent Python's `random` library is not a cryptographically secure random number generator:

> *Passeo is an open source python password generator. Versions prior to 1.0.5 rely on the python 'random' library for random value selection. The python 'random' library warns that it should not be used for security purposes due to its reliance on a non-cryptographically secure random number generator. As a result a motivated attacker may be able to guess generated passwords. This issue has been addressed in version 1.0.5. Users are advised to upgrade. There are no known workarounds for this vulnerability.*

The first hunk (lines 1-7) implements the change described by replacing the import statement for `Random` with `Secrets`. The second hunk (lines 8-33) replaces any use of the logic for using random to generate passwords, which

```diff
@@ -1,7 +1,7 @@
-import random
 import string
 import hashlib
 import requests
+import secrets

@@ -9,24 +9,28 @@ def __init__(self):
    def generate(length, numbers=False, symbols=False, uppercase=False, lowercase=False, space=False, save=False):
        password = ''
-       if numbers:
-           password += string.digits
-       if symbols:
-           password += string.punctuation
-       if uppercase:
-           password += string.ascii_uppercase
-       if lowercase:
-           if uppercase:
-               raise ValueError('Uppercase and lowercase are both true, please make one of them false.')
-           password += string.ascii_lowercase
-       if space:
+       if numbers is True:
+           password += secrets.choice(string.digits)
+       if symbols is True:
+           password += secrets.choice(string.punctuation)
+       if lowercase and uppercase == True:
+           raise ValueError('Uppercase and lowercase are both true, please make one of them false.')
+
+       if uppercase is True:
+           password += secrets.choice(string.ascii_uppercase)
+       if lowercase is True:
+           password += secrets.choice(string.ascii_lowercase)

@@ -47,27 +51,51 @@ def strengthcheck(password):
        elif y == None:
            StrengthCheckQuiz['Pwned'] = '1/3: FAIL: An error has occurred, please try again.'
        if length < 8:
-           StrengthCheckQuiz['Length'] = '2/3: FAIL: Your password is too short, it is recommended to make it longer.'
+           StrengthCheckQuiz[
+               'Length'] = '2/3: FAIL: Your password is too short, it is recommended to make it longer.'
```

Figure 1: Hunks related to patching CVE-2022-23472.

is also implementing the change described. The third and final hunk (lines 34-40) adjusts the formatting of the code, just changing the whitespace.

The first and second hunks are part of the fix for this patch, but the third hunk is not relevant as it is just changing the whitespace. Often, import statements would also not be relevant, but in this case, since the problem is with the package itself, it is relevant. Noisy hunks like the third hunk can lead to Software Vulnerability Prediction models learning incorrect information, which we hypothesize will lead to a degradation in performance.

With our approach, we filter at the hunk-level the noisy hunks using LLMs to see if we can boost the performance of popular Software Vulnerability Prediction models.

3. Effectiveness of LLM-Based Data Filtering (RQ1)

Here, we discuss the experiments conducted to answer *RQ1: Is advanced prompting of LLMs an effective approach for filtering software patch data for improving data quality?* We begin by describing the dataset, which was chosen to ensure the high quality necessary for effectively evaluating data filtering techniques. This is followed by an overview of the metrics and techniques used. We then present the results, discuss their implications, and conduct an error analysis. Finally, we consider human-in-the-loop approaches as a potential future extension to improve the filtering process.

3.1. Datasets

For RQ1, we required a pre-filtered dataset where extraneous hunks in a commit, which are not part of the specific fix, were labeled. The Tangled Commits Dataset is manually validated by software engineers, which should ensure high quality [10]. Although it is not specifically related to vulnerabilities, it is suitable for our purposes, as bug fixes are closely related.

The choice of a bug-fix dataset is necessitated by the lack of high-quality annotated vulnerability data at the hunk level. Most existing vulnerability datasets are collected automatically using heuristics and suffer from a high degree of labeling errors. Prior studies, such as Tan et al., have demonstrated significant inaccuracies in these datasets, making them unreliable for fine-grained tasks like vulnerability detection [11]. Furthermore, these datasets do not provide annotations at the hunk level, making them unsuitable for our analysis

The Tangled Commits Dataset has also been used to train models that detect vulnerability-related code at the line level [12]. In the absence of large-scale, high-quality vulnerability datasets, several recent studies have relied on bug-fix datasets as proxies for security-related tasks [13, 14]. Several studies, such as Shin et al., Yang et al., and Lomio et al. empirically show that bug-fix datasets, when characterized by traditional software metrics such as complexity, code churn, and fault history, can predict software vulnerabilities with comparable effectiveness to vulnerability-specific datasets [15, 16, 17].

3.1.1. Tangled Commits Dataset

This dataset consists of changes from code hunks containing bug-fixes manually labeled at the line-level by individuals who either had an undergraduate degree in Computer Science or a related field or proven proficiency in Java programming. Manual annotation is likely to result in a higher quality data source than automated data curation, which often relies on simple heuristics that invariable introduce noise [18, 19, 20]. The current version of the dataset is a sample of a larger dataset, SmartSHARK MongoDB Release 2.2, which uses 98 Java projects. This dataset is substantial with a plethora of information, but we specifically were looking for manually labeled hunks with issue descriptions [10]. Therefore, we randomly sampled up to 40 hunks from each project that had non-empty labels and issue IDs, sometimes sampling fewer than 40 when fewer hunks were available. We further filtered these to include only hunks with issue IDs that contained an issue description. After applying these filters, we ultimately sampled hunks from a set of 29 projects, resulting in a total of 1,093 hunks. If any line in a hunk is labeled as a bug-fix, we label the entire hunk as a bug-fix; otherwise, we label it as not a bug-fix. In our final sample, the data was relatively balanced with 507 hunks labeled as bug-fixes, and 586 were labeled as not bug-fixes.

3.2. Metrics

We formulate whether to filter out a hunk as a binary classification problem, i.e., whether a hunk contains (at least one of) the changes that fix the bug. We use metrics popular metrics for binary classification: *Precision*, *Recall*, *F1-score* and *Accuracy*, which we can compute beginning from a confusion matrix obtained from using a decision threshold. The confusion matrix consists of True Positives (TP), False Positives (FP), False Negatives (FN), and False Positives (FP).

3.2.1. Precision

Precision is the ratio of true positive observations to the total predicted positive observations.

$$Precision = \frac{TP}{TP + FP}$$

3.2.2. Recall

Recall is the ratio of true positive observations to all observations in the class yes.

$$Recall = \frac{TP}{TP + FN}$$

3.2.3. F1-Score

F1-score is the harmonic mean of Precision and Recall.

$$\text{F1-score} = 2 * \frac{Recall * Precision}{Recall + Precision}$$

Since Precision and Recall are decision-threshold dependent, we can obtain superior Precision or Recall by selecting a decision threshold often at the cost of the other. F1-score presents a more faithful representation of the classifier's performance as it counters this manipulation.

3.2.4. Accuracy

Accuracy is the ratio of true positive and true negatives to total predictions.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

3.3. LLMs

For this research, we chose the most popular open source and closed source generative LLMs available, at the time of writing: Llama3, GPT-4, and CodeLlama.

3.3.1. Llama3

Llama3 is a free, open source transformer model developed by Meta that uses decoders. It was trained on over 15 trillion tokens from various public sources. The training data for Llama3 has about four times as much code as Llama2. We utilize the 70B parameter version of this model with a temperature of 0.

3.3.2. GPT-4

GPT-4 is a paid, closed source transformer model developed by OpenAI that also uses decoders. It was trained on roughly 13 trillion tokens. As it is closed source, the specific sources of training data are not known to the public. The specific version of this model we used for our experiments was GPT-4o. We used a temperature of 0 for this model, as well.

3.3.3. CodeLlama

CodeLlama is a free, open source transformer model developed by Meta that also uses decoders. It is based on Llama2 and trained with the task of code infilling, among other tasks depending on the variant of CodeLlama. For our experiments, we utilize the 70B version of CodeLlama with a temperature of 0.

3.3.4. DeepSeek-R1

DeepSeek-R1 is a free, open source transformer model developed by DeepSeek, which is based on DeepSeek-V3-Base. We opt to use this model instead DeepSeek-Coder-V2, which targets coding tasks, due to its better performance on LiveCodeBench [21], an evaluation benchmark for multiple code tasks, including self-repair and code generation. We specifically use DeepSeek-R1-Distill-Qwen-32B due to hardware constraints.

3.4. Techniques

Prompting techniques provide a structured approach for querying an LLM, and they have been shown to significantly impact model performance across tasks. In our experiments, we applied several prompting techniques. To ensure alignment with our dataset, we adjusted the prompts based on the specific problem domain, using "bug" or "vulnerability" terminology as appropriate. This adjustment reflects the use of the Tangled Commits Dataset, which, as previously discussed, provides labeled bug hunks.

3.4.1. Zero-shot Prompting

The simplest and most widely used prompt type is the zero-shot prompt, where no examples are included in the prompt. A straightforward question, such as “Is this a software bug or vulnerability?” exemplifies a zero-shot prompt. Our zero-shot prompt structure partially incorporates the instructions given to the human annotators of the Tangled Commits Dataset. The specific zero-shot prompt used in our experiment is as follows:

Bug description: *<insert description>* The following code hunk is part of a larger commit intended to fix the above [bug/vulnerability]. Code hunks that only contain changes to whitespace, documentation, tests are not [bug-fixes/vulnerability fixes]. Code hunks that perform refactoring or unrelated changes do not qualify as [bug-fixes/vulnerability fixes].

Evaluate the code hunk below and determine if it contains a fix to the described [bug/vulnerability]: *<insert hunk>*

Please respond with "yes" or "no" only. Do not provide any more information or explanations. Format your response as follows: {"ans":"<Answer>"}

3.4.2. Few-shot Prompting

A popular type of prompting, few-shot prompting involves providing a number of labeled examples to indicate the desired output and its format. For our task, we included one example for each common change type, i.e., tests, whitespace, comments, refactoring, unrelated changes, and bug/vulnerability-fixes. Each example included a description and corresponding label. A listing of our few-shot prompt structure is shown below.

The following code hunk is part of a larger commit intended to fix the above bug. Code hunks that only contain changes to whitespace, documentation, tests are not bug fixes. Code hunks that perform refactoring or unrelated changes do not qualify as bugfixes.
Evaluate the code hunk below and determine if it contains a fix to the described bug. Please respond with "yes" or "no" only. Do not provide any more information or explanations. Format your response as follows: {"ans":"<Answer>"}

Description: *<example description 1>*
Hunk: *<example hunk 1>*
<example label 1>
Description: *<example description 2>*
Hunk: *<example hunk 2>*
<example label 2>
...
Description: *<example description 6>*

Hunk: *<example hunk 6>*
<example label 6>
Description: *<insert description>*
Hunk: *<insert hunk>*

3.4.3. Chain-of-Thought Prompting

Chain-of-Thought (CoT) prompting traditionally employs few-shot prompting to illustrate examples of sequential reasoning. However, CoT prompting can also be implemented through alternative approaches, including zero-shot prompting. The most straightforward zero-shot CoT approach involves appending the phrase "Let's think step-by-step" to the end of the prompt, a technique shown to outperform standard zero-shot prompts [22].

In this paper, we utilize a variant of zero-shot CoT prompting by requesting a summary of changes alongside a label, thereby guiding the model to exhibit a sequential reasoning process similar to "showing its work". We chose a zero-shot variant of CoT, inspired by prompting strategies described in a recent blog post on prompting [23]. The specific prompt we used for our study is shown below.

The following code hunk is part of a larger commit intended to fix the above bug. Code hunks that only contain changes to whitespace, documentation, tests are not bug fixes. Code hunks that perform refactoring or unrelated changes do not qualify as bugfixes. Please provide a summary of the changes that were implemented within the hunk.

Based on the summary, answer with yes or no whether it is relevant to fixing a bug. Provide your response in json format: {"summary":"<Summary>","ans":"<Answer>"} Hunk: *<insert hunk>*

3.4.4. Generated Knowledge Prompting

Generated Knowledge Prompting leverages few-shot prompting to produce relevant knowledge based on specific inputs. This generated knowledge is subsequently applied to a target question. To facilitate knowledge generation, we use the same examples provided in the few-shot prompt, each paired with manually generated knowledge we crafted from these examples. The knowledge structure first details specific syntactic modifications, followed by an explanation of the classification or impact of the change. For example, one instance of manually generated knowledge is: *"This change renames the variable spyStream to spy. This reflects a variable renaming, categorizing it as a refactoring code change."*

In accordance with our approach, we use few-shot prompting to guide the Large Language Model (LLM) in generating knowledge for the hunk we aim to label. We repeat this generation process three times, each iteration producing a unique knowledge set. Then, we prompt the model three additional times, incorporating one of the generated knowledge instances, a description of the problem the hunk addresses, and the hunk itself. This process outputs a label and a confidence score for each iteration, resulting in three label-score pairs. We select the label with the highest confidence score. Listings of our prompt structures for both the knowledge generation and answer generation steps are provided below.

```
Your job is to generate Knowledge for a given Hunk using the Description.
Description: <example description 1>
Hunk: <example hunk 1>
Knowledge: <example knowledge 1>
Description: <example description 2>
Hunk: <example hunk 2>
Knowledge: <example knowledge 2>
...
Description: <example description 6>
Hunk: <example hunk 6>
Knowledge: <example knowledge 6>
Description: <insert description>
Hunk: <insert hunk>
Knowledge:
```

```
Using the knowledge provided, answer the question given.
The following code hunk is part of a larger commit intended to fix the
following [bug/vulnerability]. Code hunks that only contain changes to
whitespace, documentation, tests are not [bug-fixes/vulnerability fixes].
Code hunks that perform refactoring or unrelated changes do not qual-
```

ify as [bug-fixes/vulnerability fixes]. Evaluate the code hunk below and determine if it contains a fix to the described [bug/vulnerability].
Description: *<insert description>*
Knowledge: *<insert knowledge>*
Hunk: *<insert hunk>*
Please respond with "yes" or "no" and a confidence score on a scale from 0 to 1. Do not provide any more information or explanations. Format your response as follows: {"ans":<Answer>, "conf":<Confidence Score>}

3.5. Results

To address RQ1, we conducted an evaluation of the prompting techniques (Generated Knowledge, Zero-shot, Few-shot, and Chain-of-Thought) across the selected models: Llama3, GPT4, CodeLlama, and DeepSeek-R1. The labels from the Tangled Commits Dataset, specifically at the hunk level, served as the ground truth for these evaluations. For each combination of model and prompting technique, we calculated the precision, recall, F1 score, and accuracy, with the results presented in Table 1.

Across the models, Generated Knowledge Prompting yielded the highest scores in most metrics, demonstrating superior effectiveness in capturing relevant features for classification. For both Llama3 and GPT4, Generated Knowledge Prompting achieved the highest accuracy (0.75 for Llama3 and 0.74 for GPT4) and the highest F1 score, showcasing its capability in these models to generalize well to unseen data. In GPT4, Generated Knowledge also achieved the highest recall at 0.77, indicating a strong capacity for identifying true positive cases within this model.

Interestingly, CodeLlama's performance deviated from this pattern, where Chain-of-Thought (CoT) Prompting outperformed Generated Knowledge Prompting in recall (0.85) and F1 score (0.60). This suggests that CoT prompting is better suited to CodeLlama's training objectives. However, CodeLlama's overall weak performance, despite being optimized for code-related tasks, indicates that code infilling pretraining may not translate well to tasks requiring reasoning over patch relevance.

For DeepSeek-R1, zero-shot prompting performed best, achieving the highest accuracy (0.67) and F1 score (0.58). In contrast, Generated Knowledge Prompting underperformed, with an accuracy of 0.53 and an F1 score of 0.28. This suggests that DeepSeek-R1 performs somewhat better with direct classification rather than knowledge-enhanced reasoning. This aligns with

Table 1: RQ1 Results

Model	**Prompting**	**Precision**	**Recall**	**F1**	**Accuracy**
Llama3	Gen. Knowledge	**0.81**	0.60	**0.69**	**0.75**
	Zero-shot	0.71	0.52	0.60	0.68
	Few-shot	0.83	0.23	0.37	0.62
	COT	0.63	**0.65**	0.64	0.66
GPT4	Gen. Knowledge	0.71	**0.77**	**0.73**	**0.74**
	Zero-shot	**0.81**	0.51	0.62	0.71
	Few-shot	0.69	0.61	0.65	0.69
	COT	0.62	0.64	0.63	0.65
CodeLlama	Gen. Knowledge	**0.67**	0.00	0.01	**0.54**
	Zero-shot	0.32	0.04	0.07	0.52
	Few-shot	0.53	0.17	0.26	0.54
	COT	0.46	**0.85**	**0.60**	0.47
DeepSeek-R1	Gen. Knowledge	0.49	0.20	0.28	0.53
	Zero-shot	**0.71**	**0.48**	**0.58**	**0.67**
	Few-shot	0.00	0.00	0.00	0.54
	COT	0.57	0.40	0.47	0.58

DeepSeek-R1's "mixture of experts" approach, activating specific subsets of its parameters based on the input, which allows for efficient adaptability to various tasks in a zero-shot setting.

3.6. Implications

Out of all the LLMs, CodeLlama and DeepSeek-R1 performed the poorest for this task, overall. CodeLlama is based on Llama2, which is not trained on as much code as Llama3, which explains why Llama3 performed better. Additionally, CodeLlama is fine-tuned with the task of code infilling, which may not be related enough to the task of identifying bug-fixes. DeepSeek-R1 was primarily intended for tasks requiring logical inference, mathematical reasoning, and real-time problem-solving. The type of code classification task the is the focus of this paper is likely out of its trained expertise.

Generated Knowledge Prompting is typically used in the domain of commonsense reasoning. In the original study [7], Generated Knowledge Prompting is compared against vanilla prompting, random knowledge, related knowledge, self-talk [24], retrieval-based knowledge, and few-shot prompting. In contrast, Chain-of-Thought Prompting [25] has been applied to a variety of tasks, most relevantly to math word problems and commonsense reasoning, and it is typically compared against vanilla prompting. Both of the original studies on Chain-of-Thought Prompting and Generated Knowledge Prompting conduct experiments on the Commonsense QA dataset [26]. However, because the models used in these experiments are different, the results and gains are not directly comparable. It is noted, however, that Chain-of-Thought Prompting results show less improvement on this dataset compared to others, such as math word problem datasets. This suggests that Generated Knowledge Prompting may be more suitable for qualitative tasks, whereas Chain-of-Thought Prompting may better serve quantitative tasks. Our task, between the two, leans towards the qualitative side, so Generated Knowledge is more appropriate, explaining its higher results compared to Chain-of-Thought Prompting.

While our best-performing configuration (GPT-4 with generated knowledge prompting) achieves 75% accuracy and 0.73 F1 score on hunk-level filtering, these results are competitive with state-of-the-art systems. For instance, ActiveClean, a recent traditional ML-based approach for line-level patch filtering, reports F1 scores of 70–74% on Java datasets and 70.23% on C code, but it requires explicit feature engineering and iterative labeling via active learning [12]. Our approach achieves similar results without handcrafted features and with minimal supervision.

Overall, these findings imply that for tasks requiring contextual understanding within code—such as bug fix classification or code refactoring detection—Generated Knowledge Prompting may provide more accurate results, especially with models like Llama3 and GPT4 that are pre-trained on diverse data.

3.7. Error Analysis

To understand what are the types of hunks that are still difficult to classify as bug-fix related, we investigate what types of errors that were made by the best performing configuration: Generated Knowledge Prompting with GPT-4. To this end, we randomly sampled 30 misclassified hunks from each of the false positive and false negative groups, i.e., a total of 60 hunks. The first author conducted a systematic analysis of each hunk to determine the likely root causes for the error. In cases where the classification was unclear or involved a borderline case, the author discussed the instance with another author until both reached an agreement. Both of the authors have advanced training in computer science, including expertise in secure programming.

Of the 30 false negatives, all but one of the errors were due to incorrect annotations in the dataset. Several of these, labeled as bug-fixes, were actually whitespace changes, unrelated modifications, or documentation updates. A common annotation error was labeling the addition of a blank line as a bug-fix. While most of these incorrect annotations were among the 30 false negatives, one was found in the 30 false positives. This false positive involved a hunk labeled as non-bug-fix, but it actually included part of a fix for a concurrency issue. Specifically, it addressed a `ConcurrentModificationException` that occurred multiple threads sharing a Java class loader, which should have been supported. The reason for these misannotations could be that approximately 7.9% of the labeled lines in the Tangled Commits dataset were unintentionally mislabeled and 14.3% of production code lines had no consensus among annotators [10].

The next most common error in 19 of the 60 samples was with unrelated changes, i.e., changes that did not fix a bug in the code, among the false positives. This is expected as unrelated changes often bring functional changes, which can be difficult to distinguish from bug-fixes. One example of an unrelated change that was misclassified was the following:

```
+            if (this == obj) return true;
```

The description for this explained that *"...the workaround is to provide a comparator"*. However, the change above is unrelated to this description.

Another phenomenon we observed was GPT-4 identifying another issue in the code that was fixed in the patch. This did not happen often, only occurring in 4 out of the 60 samples, among the false positives. We can find these occurrences by reading the generated knowledge. For example:

> *Knowledge: This hunk modifies the behavior of a map implementation to correctly track the "dirty" state of its entries. Previously, the code would mark an entry as dirty whenever a new value was put into the map, regardless of whether the new value was different from the old value. The updated code removes the key from the 'keyStates' map first, then checks if the new value is different from the old value. If they are different (or if the old value is null), it marks the entry as dirty. This change ensures*

> *that only genuinely modified entries are marked as dirty, improving performance by reducing unnecessary dirty state tracking.*

This knowledge appears to be talking about a hunk that is a bug-fix, but the description is about leaking input streams, so in reality this is a unique case of an unrelated change, where the change is an unrelated bug-fix.

Another uncommon issue was with the descriptions provided to the model. In 2 out of the 60 samples, from the false positives, the description did not provide enough relevant details to be useful to the LLM. For example:

> *During reading the Ivy source code, I stumbled on this possible issue. Please, see the attached patch.*

Descriptions like these provide no input to the LLM as to what the issue is and can lead to unrelated changes being labeled as bug-fixes.

There were very few mistakes regarding refactoring. This is easier to identify compared to unrelated changes, so logically it follows that there would be only a small amount of mistakes with this compared to unrelated changes. Out of the 60 instances of mistakes, only two were refactoring changes labeled as bug-fixes. A common example of refactoring is updating code to reflect a change in name of a variable or function.

3.8. Human-in-the-Loop Dataset Curation

One avenue for improving software patch filtering is through joint human and LLM filtering. Prior studies have explored annotation schemes that combine automated approaches with some level of human involvement. Human-in-the-loop strategies can enhance predictive performance [12, 27] and improve explainability [28]. For instance, INSPECTOR uses assistive labeling to produce various scores for rating automatically generated annotations, which human annotators can manually verify when scores fall below a certain threshold [29]. Similarly, our approach generates confidence scores for final labels. Setting a threshold for confidence scores could ensure that labels with lower confidence are verified by a human annotator, potentially increasing the reliability of the filtering process.

4. Impact of Cleaned Data on Vulnerability Prediction (RQ2)

Next, we discuss the experiments conducted to answer *RQ2: Does filtered software vulnerability patch data improve the performance of automated* *software vulnerability prediction?* We start by describing the BigVul dataset, which is one of the most popular patch datasets available today. This is followed by an outline of the metrics and the recent popular Software Vulnerability Prediction models we experimented with. We then present our findings, discuss their implications, and perform an error analysis.

4.1. Dataset

For RQ2, we require a large-scale dataset focused on software vulnerabilities to evaluate the impact of data cleaning on software vulnerability prediction. We use the popular software vulnerabilities dataset BigVul [30], which contains vulnerable and non-vulnerable (clean) functions extracted from the NVD, to assess the efficacy of our chosen filtering method for training a prediction model.

The dataset is split into five folds, ensuring that functions from the same commit do not appear across different folds. We prepare two versions of the dataset: (1) a filtered version using Generative Knowledge Prompting (our best-performing method from RQ1) and (2) an unfiltered version. We train software vulnerability prediction models on four of the filtered or unfiltered folds and evaluate on a held-out, filtered test set.

4.2. Methodology

Using the aforementioned BigVul dataset, we experiement with training different popular software vulnerability prediction models on the filtered and unfiltered data. Specifically, we chose LineVul [31], CodeBERT [32], and CodeT5 [33]. While LineVul is a technique that is specific for software vulnerability prediction, CodeBERT and CodeT5 are general language models for source code. Therefore, CodeBERT and CodeT5's configuration in our experiments, as described below, follows that of recent research on software vulnerability prediction [18]; we train both models for 10 epochs with a learning rate of 2×10^{-7}.

4.2.1. LineVul

LineVul is a deep neural network based model that is based on CodeBERT, a pre-trained model specifically designed for programming languages. LineVul first identifies vulnerable functions and then identifies specific lines that are vulnerable using the underlying language model's attention mechanism [34]. LineVul was recently evaluated as one of the best-performing models among various vulnerability prediction approaches [35]. We use the default parameters for LineVul, without setting the random seed, and train for 10 epochs. We use the validation set F1 to pick the best checkpoint to compute the metrics on the test set.

4.2.2. CodeBERT

A deep neural network based model that is designed for programming languages. The architecture of CodeBERT is the same transformer-based architecture as RoBERTa-base [36]. It is pre-trained on CodeSearchNet [37], a bimodal and unimodal dataset of code and natural language. There are two training objectives used. The first is Masked Language Modeling (MLM), which where some individual tokens are masked and the goal is to predict what was masked. This first objective is done with only the bimodal data. The second is Replaced Token Detection (RTD). This objective is similar to the MLM objective, but instead of just masking, it replaces the masked input with some corrupt input, generated by a generator. To do this, it uses both the unimodal and bimodal portions of the dataset. We add a classification head, consisting of two linear layers, on top of this model to map the generated representations to a label: vulnerable or not vulnerable.

4.2.3. CodeT5

CodeT5 is a deep neural network based model that is similar to CodeBERT in the sense that it is also a pre-trained model designed for programming languages, but the underlying model here is T5, and encoder-decoder based transformer model, whereas CodeBERT is encoder-only. CodeT5 is pre-trained on CodeSearchNet [37]. During training, it is introduced to both code and a combination of code and natural language. The training tasks used are Masked Span Prediction (MSP), Identifier Tagging (IT), and Masked Identifier Prediction (MIP). MSP is similar to MLM, but in this case, sequences or spans of tokens are masked. IT is, as the name implies, predicting whether a code token is an identifier or not. Finally, MIP is similar to MLM as well, except only identifier tokens are masked. We again add a classification head, consisting of two linear layers, on top of this model to map the generated representations to a label: vulnerable or not vulnerable.

4.3. Metrics

We will use the metrics from RQ1 (i.e., precision, recall, F1-score, and accuracy) again in RQ2. For details on these metrics, refer to Sections 3.2.1, 3.2.2, 3.2.3, and 3.2.4. We also leverage two additional metrics aimed at imbalanced data.

4.3.1. Matthews Correlation Coefficient (MCC)

MCC measures the correlation between predicted and actual values.

$$\text{MCC} = \frac{TN \cdot TP - FN \cdot FP}{\sqrt{(TP + FP) \cdot (TP + FN) \cdot (TN + FP) \cdot (TN + FN)}}$$

4.3.2. AUC-ROC

AUC-ROC is the area under the receiver operator characteristic curve. The curve itself shows the tradeoff between the True Positive Rate (TPR) and the False Positive Rate (FPR), and the area under this curve is a single numerical value to represent this. 1.0 represents perfect classification, and 0.5 is at the level of random guessing. Because it considers TPR and FPR, it is better than accuracy for problems with class imbalances, like ours. TPR is the same as recall, defined in 3.2.2, FPR is defined as follows.

$$\text{FPR} = \frac{FP}{FP + TN}$$

4.3.3. AUC-PR

AUC-PR is similar to AUC-ROC, in that it is also area under the curve. In this case, instead of measuring the tradeoff between FPR and TPR, we measure the tradeoff between precision and recall. This is useful in the case where datasets are imbalanced such that the majority class is the negative class. Precision is defined earlier in 3.2.1, and recall is defined earlier in 3.2.2.

P-values are calculated using the Mann-Whitney U-test, which we use because our data is not normally distributed, but the shape of the distribution of the data is similar. This test ranks all data points, then, using this, a Mann-Whitney U-test statistic is calculated.

4.4. Results

For this research question, we evaluate the model's performance with precision, recall, F1, MCC, Accuracy, AUC-ROC, and AUC-PR. In general, we note that the most sophisticated method, LineVul [31], performs higher than the other models we experiment with. The difference in accuracy is smaller than the difference in other metrics.

Most notably, LineVul trained on the unfiltered and filtered data performs at an average precision of 0.668 and 0.715 respectively, whereas CodeBERT performs at 0.475 and 0.511, and CodeT5 performs at 0.532 and 0.578. Also of note is the difference in terms of F1. LineVul performs at a 0.754 average for both versions. CodeBERT performs at 0.556 and 0.573 in terms of F1, and CodeT5 performs at a 0.594 and 0.616 in terms of F1. Performance for these two models is lower than the reported results on the original BigVul dataset in the PrimeVul study [18]; however we notice that CodeT5 still performs at a slightly higher F1 than CodeBERT.

For LineVul specifically, there is a significant increase in precision when training using the filtered datasest (+7.19%). With this comes a significant drop in recall (-7.96%). Although, the drop in recall is lower on CodeBERT (-2.89%) and CodeT5 (-3.17%) when compared against LineVul. Because the drop is lower, the F1 increases slightly for those two models, with CodeBERT increasing by 3.06% and CodeT5 increasing by 3.70%; however, the increase in for CodeBERT is statistically insignificant. For LineVul, it remains the same. MCC decreases slightly with the filtered LineVul model (-1.07%), but for the other filtered models it increases slightly. Specifically, CodeBERT increases by 3.80% in MCC, and CodeT5 increases by 4.53% in MCC. Accuracy slightly increases for LineVul, CodeBERT, and CodeT5 with increases of 0.20%, 0.63%, 0.62% respectively, with the increase in the LineVul model being statistically insignificant; conversely, AUC-ROC slightly decreases for CodeBERT and CodeT5 by 0.47% and 0.12% respectively, but these values are not statistically significant. There is a 9.25% gain in AUC-PR for LineVul, but for the other models it is much less, at 2.16% for CodeBERT and 1.42% for CodeT5. Overall, the largest gains we observe are in precision, and the largest losses we see are in recall, to a lesser degree.

For CodeBERT, the training times showed a slight reduction when using the filtered dataset. On average, training on the unfiltered dataset took 35,807 $\pm$ 972 seconds, while training on the filtered dataset required 35,097 $\pm$ 1,034 seconds. This corresponds to a 1.98% decrease in training time. Notably, this reduction occurs despite the dataset size remaining unchanged, suggesting that correcting the labels of hunks leads to a slightly more efficient learning process for vulnerability prediction models.

4.5. Implications

We observe the biggest gains in precision and the biggest drop in recall. This leads to an overall slight increase in F1, which is different from what was observed in the PrimeVul study [18], where the F1 significantly dropped on the PrimeVul dataset due to the original dataset's quality and exact copies being present, which inflated performance.

Table 2: Comparison of Software Vulnerability Prediction models using Filtered and Unfiltered training data.

Model	Training Data	Precision	Recall	F1	MCC	Accuracy	AUC-ROC	AUC-PR
LineVul	Unfiltered	0.668 ± 0.033	0.867 ± 0.025	0.754 ± 0.023	0.748 ± 0.021	0.976 ± 0.003	0.948 ± 0.017	0.692 ± 0.030
	Filtered	0.716 ± 0.033	0.798 ± 0.033	0.754 ± 0.019	0.740 ± 0.019	0.978 ± 0.003	0.948 ± 0.014	0.756 ± 0.031
	+/-	+7.19%	-7.96%	0.00%	-1.07%	+0.20%	0.00%	+9.25%
	p-value	$5.9e-05$	$6.3e-11$	$8.40e-02$	$3.27e-02$	$1.09e-01$	$1.24e-01$	$1.92e-08$
CodeBERT	Unfiltered	0.475 ± 0.027	0.761 ± 0.024	0.556 ± 0.023	0.658 ± 0.024	0.955 ± 0.005	0.844 ± 0.019	0.835 ± 0.005
	Filtered	0.511 ± 0.038	0.739 ± 0.023	0.573 ± 0.028	0.683 ± 0.023	0.961 ± 0.005	0.84 ± 0.02	0.853 ± 0.011
	+/-	+7.58%	-2.89%	+3.06%	+3.80%	+0.63%	-0.47%	+2.16%
	p-value	$1.67e-3$	$1.45e-2$	$5.02e-2$	$1.96e-3$	$3.16e-3$	$2.67e-1$	$1.13e-4$
CodeT5	Unfiltered	0.532 ± 0.022	0.756 ± 0.027	0.594 ± 0.017	0.706 ± 0.014	0.964 ± 0.004	0.842 ± 0.02	0.846 ± 0.006
	Filtered	0.578 ± 0.022	0.732 ± 0.028	0.616 ± 0.014	0.738 ± 0.014	0.97 ± 0.003	0.841 ± 0.02	0.858 ± 0.01
	+/-	+8.65%	-3.17%	+3.70%	+4.53%	+0.62%	-0.12%	+1.42%
	p-value	$1.18e-5$	$1.86e-2$	$1.41e-3$	$3.51e-6$	$1.41e-4$	$9.83e-1$	$4.27e-3$

By relabeling the mislabeled functions using our approach, we are effectively teaching the models to be more selective than before, which naturally leads to an increase in precision. A natural tradeoff with precision going up is recall going down. We attribute the drop in recall to the fact that filtering significantly reduced the proportion of vulnerable functions in the dataset from 6.2% to 4.4%. This approximately 29% reduction makes the data even more imbalanced, which may have caused the trained vulnerability prediction models to become overly conservative. If the models become more selective, they are more prone to missing vulnerabilities. The gains in precision being greater than the drop in recall suggests that the filtering accomplishes more than reducing the number of positives; it adds to the quality of the dataset. If it was just reducing the number of positives, then the precision would not outweigh the recall, and the F1 would not increase.

The implications of our approach extend to improving vulnerability dataset quality while reducing reliance on costly and non-scalable manual validation. Public vulnerability sources, such as the National Vulnerability Database (NVD), often contain incomplete or incorrect information [11, 38]. Manual validation improves data quality but is slow, expensive, and requires familiarity with different software projects [39]. LLM-based patch filtering can remove mislabeled or extraneous patches, improving the reliability of vulnerability prediction models while reducing manual effort. By leveraging contextual reasoning, we observe that our approach improves precision without significantly reducing relevant data, leading to improvements vulnerability prediction model performance. This introduces and approach for security teams to curate high-quality datasets at scale, improve automated vulnerability detection, and reduce false positives, allowing them to address real security threats more effectively.

4.6. Error Analysis

To understand the types of errors that the models trained on the filtered dataset make compared to the models trained on the unfiltered dataset, we examine the false negatives and false positives predicted from the models trained on the filtered dataset. Specifically, we look at cases where these predictions differed from the models trained on the unfiltered dataset. We randomly sample 30 false positives and 30 false negatives, i.e. a total of 60 samples from one of the folds from the LineVul experiments since this model performed higher than the other models as observed earlier. When classifying the reasons for the error, when provided, we analyze the summary, commit message, and the specific changes to the code.

Among the 30 false negatives, we found only 13 actual errors in prediction. False negatives in this case are when the filtered model predicts not vulnerable and the ground truth is vulnerable. The types of these errors varied greatly, but some examples include overflows, heap corruption, etc. The rest of the 17 errors with the false negatives involved incorrect original annotation, that was not able to be filtered with our method.

One very common type of annotation error was unrelated changes being labeled as positives in the original annotation, making up 13 out of the 30 false negatives. These changes often included the addition of features, not fixes to vulnerabilities. An example is the following:

```
void PaletteTool::RegisterToolInstances(PaletteToolManager* tool_manager) {
   tool_manager->AddTool(base::MakeUnique<CaptureRegionAction>(tool_manager)
      );
   tool_manager->AddTool(base::MakeUnique<CaptureScreenAction>(tool_manager)
      );
   tool_manager->AddTool(base::MakeUnique<CreateNoteAction>(tool_manager));
+  tool_manager->AddTool(base::MakeUnique<MagnifierMode>(tool_manager));
}
```

In this example from Chromium, this function is a part of adding a partial magnifier to the software. This is adding a feature, not fixing a software vulnerability, so it is an unrelated change.

Another case we observed was bugfixes. These differ from vulnerabilities as they do not inherently cause a security risk. There were only four cases of bugfixes in the false negatives. These required us to examine both the commit messages and code. The commit messages, particularly, were very indicative of whether a change was bugfix or not. An example of such a commit message is the following:

Make NotifyHeadersComplete the last call in the function.
BUG=82903 ...

With regards to the false positives, all 30 sampled were all errors with the predictions, not the dataset. Upon examining each sample's function, we found these functions were unchanged in the patch. With our approach, we filter the positive samples and, when applicable, relabel them to negative. Interestingly, upon further investigation, it appears that all 30 of these samples are labeled from the original annotation, not from our filtering of the positive labeled samples.

5. Related Work

We organize the research related to this paper into four different categories: 1) vulnerability data curation and filtering techniques; 2) recent software vulnerability prediction techniques; 3) LLMs for Software Engineering tasks; and 4) applications of prompting to related tasks.

The majority of vulnerability patch datasets directly curate data from the National Vulnerability Database (NVD) or similar public sources. The research community has diligently contributed to curating multiple vulnerability patch datasets [40, 41, 42, 43, 44, 45, 46, 20, 47]. However, the current set of curated datasets do not limit the noise and error present in the NVD. Therefore, the quality of vulnerability patch datasets is a pressing concern. One way to enhance dataset quality is through manual curation, as exemplified by the SAP dataset, which is curated by developer teams at SAP [46]. However, manual methods face scalability issues due to the extensive manual review required. Manual patch review is time-consuming even for software security experts as they are usually unfamiliar with the specific software projects that are patched. An alternative to enhancing dataset quality without the scalability issues presented by manual curation is to filter the datasets using semi-automated curation techniques. An example of this is using active learning [12, 27] to identify key instances to manually label. There are also fully-automated curation techniques, typically also using deep learning [48, 19, 49, 50].

The field of software vulnerability prediction addresses the demand for detecting and/or classifying vulnerabilities before they are discovered and exploited. Software vulnerability prediction tends to follow trends and developments in Natural Language Processing, as code is ultimately text. Thus, modern software vulnerability prediction models predominantly use deep learning. There are a number of different architectures that are used for software vulnerability prediction, including Convolutional Neural Networks (CNNs) [51, 52, 53], Recurrent Neural Networks (RNNs) [54, 55, 56, 57], Graph Neural Networks (GNNs) [58, 59, 60, 61], and Transformers [31, 62].

Large Language Models (LLMs) are increasingly applied across diverse software engineering tasks, offering new capabilities for automation. In requirements engineering, LLMs support use cases such as eliciting, analyzing, and even partially generating software requirements [63]. In code generation, they assist in producing boilerplate code, implementing common patterns, and accelerating prototyping workflows [64]. For program repair, LLMs are used to generate candidate patches, fix common bugs, and suggest alternatives informed by training data [65]. In test case generation, LLMs help by creating unit and integration tests from function signatures or docstrings, enabling broader coverage with minimal manual input [66]. They are also applied in defect prediction to analyze historical code and commit patterns to forecast likely fault-prone components [67]. In vulnerability prediction, LLMs have been leveraged to detect potential security flaws in codebases, enhancing traditional static analysis methods [18]. Despite this range of applications, effective deployment of LLMs still faces challenges in reliability, alignment with developer intent, and the need for verification of their output.

Deep learning methods are data-hungry, and large amounts of reliable data are not always available for software vulnerability prediction and other related fields. Generative LLMs like Llama and GPT provide an alternative by using prompting to extract relevant patterns and provide additional context. Some examples of where prompting is used in relation to software vulnerability tasks include software vulnerability prediction and software vulnerability repair. Prompting for software vulnerability prediction varies in approach. Chain-of-Thought prompting is very popular among advanced prompting techniques. This is used both in the zero-shot form [68, 69] and in the few-shot form [70]. Another approach implemented by some is to prompt-tune, as opposed to manually choosing prompts and examples [71] For vulnerability repair using prompting, once again Chain-of-Thought prompting is used [70] along with specialized zero-shot prompting [72]. Recently, Yu Nong et al. introduced LLMPatch, an automated patch generation system that harnesses pre-trained LLMs and adaptive chain-of-thought prompting to fix vulnerabilities [73]. Similarly, VRpilot uses a chain-of-thought prompt for automatic vulnerability repair. The LLM reasons about vulnerable code, and then iteratively refines the patch based on external feedback (compiler errors, sanitizer outputs, test results) [74].

Prompting has been used for a few related software vulnerability related tasks, but not extensively and not for data curation on software vulnerability datasets specifically. Our work aims to address this gap by investigating the potential of using different prompting techniques and LLMs for filtering noisy data, and ultimately, identifying the impact of said filtering on the performance of a Deep Learning model on the task of software vulnerability prediction.

6. Threats to Validity

LLMs have many hyperparameters that can introduce bias into experiments, creating a threat to internal validity. Particularly, there is one hyperparameter known as temperature which changes the randomness or creativity of responses. To mitigate this, we set the temperature to 0 to reduce variance as much as possible. For Generated Knowledge Prompting in RQ1, we generate knowledge three times, use each knowledge to generate a label and confidence score, and pick the label with the highest confidence. This way, we address the variance in the generated knowledge.

In RQ1, the best and chosen approach to use on RQ2 had an accuracy of 0.74, which suggests that the filtration may worsen the quality of the dataset we test with in RQ2, creating a threat to construct validity. From our error analysis in RQ1, though, we approximate that half of the errors were due to mislabeling in the dataset, so the quality of the data after filtering with our method is of higher quality than the metrics may make it seem.

Because we observe numerous instances where the data has been annotated incorrectly, it is unclear whether the predictions marked as correct are predicting on mislabeled data, introducing a threat to internal validity. However, this dataset is considered the gold standard for annotation, considering it was manually annotated. We believe that the incorrect annotations we encountered were exceptions.

For RQ1, we use a bug-fix dataset to determine the best prompting strategy for filtering. However, this is on a bug-fix dataset, not a software vulnerability dataset. The same approach on one task may not have the same performance on the other task, even if those tasks are similar, creating a threat to internal validity. We modify our prompts slightly to be more applicable to the task it is addressing, as we discuss earlier.

LLMs are known to hallucinate, which could introduce incorrect filtering decisions. To mitigate this, we use structured prompts that constrain responses and focus on factual reasoning. Additionally, LLMs are pre-trained on open-source data, potentially including open-source vulnerability datasets, which could bias their performance. While this may improve their familiarity with software patches, it also risks data leakage, which we attempt to minimize by testing across multiple LLMs with different architectures.

In RQ2, we only run our experiments on a few models. It is possible that on other models, the conclusions we make will not hold, introducing a threat to external validity. However, the results across these models were consistent in terms of performance change when training with the filtered or unfiltered dataset, which suggests that this will hold even on different models. Additionally, these models are popular for software vulnerability prediction.

Finally, the sampling of the data itself is a threat to external validity, as it cannot be confirmed that the data sampled will be representative of future vulnerabilities. The dataset we use samples across multiple different projects to mitigate the bias that would be introduced by just using one.

7. Conclusions

This paper explored the efficacy of using LLMs for improving the quality of software vulnerability patch data. To investigate this, we employed various prompting strategies to guide LLMs in filtering noisy patches, applied the most effective approach, Generated Knowledge Prompting, to the BigVul dataset, and evaluated the impact of this filtered data on the performance of three popular automated vulnerability prediction models. Our findings demonstrated that Generated Knowledge Prompting significantly enhances the precision of filtering, resulting in a 7-8% in precision (while recall drops) and marginal improvement in accuracy across popular models such as LineVul, CodeBERT, and CodeT5. Overall, our results highlight the potential of LLMs, particularly when guided by effective prompting strategies, to not only refine the quality of vulnerability patch data but also improve the reliability and precision of downstream vulnerability prediction models.

However, there are areas for further investigation. Our error analysis revealed that while precision improved, there was a slight trade-off in recall. This suggests that further refinement in LLM prompting strategies, LLM fine tuning, or integration of human-in-the-loop processes could help mitigate these limitations. Additionally, future research could explore the application of other advanced LLMs to enhance both the filtering process and the interpretative accuracy of vulnerability patches in diverse software engineering contexts. Our approach could also be extended to filter other types of noise, addressing additional data quality attributes like consistency and completeness, as highlighted by Croft et al. [39]. This study provides an initial attempt at leveraging LLMs in software vulnerability data curation, offering a promising avenue for future empirical research to refine and expand upon these findings across different datasets and prediction models.

References

[1] National Vulnerability Database, `https://nvd.nist.gov/` (2024).

[2] Snyk Vulnerability Database, `https://security.snyk.io/` (2024).

[3] GitHub Advisory Database, `https://github.com/advisories` (2024).

[4] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, K. Narasimhan, Tree of thoughts: Deliberate problem solving with large language models, Advances in Neural Information Processing Systems 36 (2024).

[5] M. Besta, N. Blach, A. Kubicek, R. Gerstenberger, M. Podstawski, L. Gianinazzi, J. Gajda, T. Lehmann, H. Niewiadomski, P. Nyczyk, et al., Graph of thoughts: Solving elaborate problems with large language models, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 38, 2024, pp. 17682–17690.

[6] L. LUO, Y.-F. Li, R. Haf, S. Pan, Reasoning on graphs: Faithful and interpretable large language model reasoning, in: The Twelfth International Conference on Learning Representations, 2024.

[7] J. Liu, A. Liu, X. Lu, S. Welleck, P. West, R. L. Bras, Y. Choi, H. Hajishirzi, Generated knowledge prompting for commonsense reasoning (2022). `arXiv:2110.08387`.
URL `https://arxiv.org/abs/2110.08387`

[8] A. Alali, H. Kagdi, J. I. Maletic, What's a typical commit? a characterization of open source software repositories, in: 2008 16th IEEE International Conference on Program Comprehension, 2008, pp. 182–191. `doi:10.1109/ICPC.2008.24`.

[9] E. W. Myers, Ano(nd) difference algorithm and its variations, Algorithmica 1 (1–4) (2023) 251–266. doi:10.1007/BF01840446.

[10] S. Herbold, A. Trautsch, B. Ledel, A. Aghamohammadi, T. A. Ghaleb, K. K. Chahal, T. Bossenmaier, B. Nagaria, P. Makedonski, M. N. Ahmadabadi, K. Szabados, H. Spieker, M. Madeja, N. Hoy, V. Lenarduzzi, S. Wang, G. Rodríguez-Pérez, R. Colomo-Palacios, R. Verdecchia, P. Singh, Y. Qin, D. Chakroborti, W. Davis, V. Walunj, H. Wu, D. Marcilio, O. Alam, A. Aldaeej, I. Amit, B. Turhan, S. Eismann, A.-K. Wickert, I. Malavolta, M. Sulir, F. Fard, A. Z. Henley, S. Kourtzanidis, E. Tuzun, C. Treude, S. M. Shamasbi, I. Pashchenko, M. Wyrich, J. Davis, A. Serebrenik, E. Albrecht, E. U. Aktas, D. Strüber, J. Erbel, Large-scale manual validation of bug fixing commits: A fine-grained analysis of tangling (2020). arXiv:2011.06244.

[11] X. Tan, Y. Zhang, C. Mi, J. Cao, K. Sun, Y. Lin, M. Yang, Locating the Security Patches for Disclosed OSS Vulnerabilities with Vulnerability-Commit Correlation Ranking, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21, 2021, pp. 3282–3299, event-place: Virtual Event, Republic of Korea.

[12] A. K. Joshy, M. S. Alam, S. Sharmin, Q. Li, W. Le, Activeclean: Generating line-level vulnerability data via active learning (2023). arXiv:2312.01588.
URL https://arxiv.org/abs/2312.01588

[13] Z. Chen, S. Kommrusch, M. Monperrus, Neural transfer learning for repairing security vulnerabilities in c code, IEEE Transactions on Software Engineering 49 (1) (2023) 147–165. doi:10.1109/TSE.2022.3147265.

[14] J. Chi, Y. Qu, T. Liu, Q. Zheng, H. Yin, SeqTrans: Automatic Vulnerability Fix Via Sequence to Sequence Learning , IEEE Transactions on Software Engineering 49 (02) (2023) 564–585. doi:10.1109/TSE.2022.3156637.
URL https://doi.ieeecomputersociety.org/10.1109/TSE.2022.3156637

[15] Y. Shin, L. Williams, Can traditional fault prediction models be used

for vulnerability prediction?, Empirical Software Engineering 18 (2013) 25–59.

[16] L. Yang, X. Li, Y. Yu, Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes, in: GLOBECOM 2017-2017 IEEE Global Communications Conference, IEEE, 2017, pp. 1–7.

[17] F. Lomio, E. Iannone, A. De Lucia, F. Palomba, V. Lenarduzzi, Just-in-time software vulnerability detection: Are we there yet?, Journal of Systems and Software 188 (2022) 111283.

[18] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, Y. Chen, Vulnerability detection with code language models: How far are we?, in: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), IEEE Computer Society, 2024, pp. 469–481.

[19] S. Wang, Y. Zhang, L. Bao, X. Xia, M. Wu, Vcmatch: A ranking-based approach for automatic security patches localization for oss vulnerabilities, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022, pp. 589–600. `doi:10.1109/SANER53432.2022.00076`.

[20] T. G. Nguyen, T. Le-Cong, H. J. Kang, X.-B. D. Le, D. Lo, VulCurator: A Vulnerability-Fixing Commit Detector (Sep. 2022). `doi:10.48550/arXiv.2209.03260`.
URL `http://arxiv.org/abs/2209.03260`

[21] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, I. Stoica, Livecodebench: Holistic and contamination free evaluation of large language models for code (2024). `arXiv:2403.07974`.
URL `https://arxiv.org/abs/2403.07974`

[22] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, Y. Iwasawa, Large language models are zero-shot reasoners (2023). `arXiv:2205.11916`.
URL `https://arxiv.org/abs/2205.11916`

[23] Z. Yan, Prompting fundamentals and how to apply them effectively, eugeneyan.com (May 2024).

[24] V. Shwartz, P. West, R. Le Bras, C. Bhagavatula, Y. Choi, Unsupervised commonsense question answering with self-talk, in: B. Webber, T. Cohn, Y. He, Y. Liu (Eds.), Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, Online, 2020, pp. 4615–4629. doi:10.18653/v1/2020.emnlp-main.373.

[25] J. Wei, X. Wang, D. Schuurmans, M. Bosma, b. ichter, F. Xia, E. Chi, Q. V. Le, D. Zhou, Chain-of-thought prompting elicits reasoning in large language models, in: S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, A. Oh (Eds.), Advances in Neural Information Processing Systems, Vol. 35, Curran Associates, Inc., 2022, pp. 24824–24837.

[26] A. Talmor, J. Herzig, N. Lourie, J. Berant, Commonsenseqa: A question answering challenge targeting commonsense knowledge (2019). arXiv:1811.00937.
URL https://arxiv.org/abs/1811.00937

[27] Z. Yu, C. Theisen, L. Williams, T. Menzies, Improving vulnerability inspection efficiency using active learning, IEEE Transactions on Software Engineering 47 (11) (2021) 2401–2420. doi:10.1109/tse.2019.2949275.

[28] I. Arous, L. Dolamic, J. Yang, A. Bhardwaj, G. Cuccu, P. Cudré-Mauroux, Marta: Leveraging human rationales for explainable text classification, Proceedings of the AAAI Conference on Artificial Intelligence 35 (7) (2021) 5868–5876. doi:10.1609/aaai.v35i7.16734.

[29] H. J. Kang, F. Harel-Canada, M. A. Gulzar, V. Peng, M. Kim, Human-in-the-loop synthetic text data inspection with provenance tracking (2024). arXiv:2404.18881.
URL https://arxiv.org/abs/2404.18881

[30] J. Fan, Y. Li, S. Wang, T. N. Nguyen, A c/c++ code vulnerability dataset with code changes and cve summaries, in: Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 508–512. doi:10.1145/3379597.3387501.

[31] M. Fu, C. Tantithamthavorn, Linevul: A transformer-based line-level vulnerability prediction, in: 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), 2022, pp. 608–620. doi:10.1145/3524842.3528452.

[32] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, Codebert: A pre-trained model for programming and natural languages (2020). arXiv:2002.08155.

[33] Y. Wang, W. Wang, S. Joty, S. C. H. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation (2021). arXiv:2109.00859.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need (2023). arXiv:1706.03762.

[35] B. Steenhoek, M. M. Rahman, R. Jiles, W. Le, An empirical study of deep learning models for vulnerability detection, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 2237–2248.

[36] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, V. Stoyanov, Roberta: A robustly optimized bert pretraining approach (2019). arXiv:1907.11692.

[37] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, M. Brockschmidt, Codesearchnet challenge: Evaluating the state of semantic code search (2020). arXiv:1909.09436.

[38] S. Wang, Y. Zhang, L. Bao, X. Xia, M. Wu, VCMatch: A ranking-based approach for automatic security patches localization for oss vulnerabilities, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, IEEE, 2022, pp. 589–600.

[39] R. Croft, M. A. Babar, M. M. Kholoosi, Data quality for software vulnerability datasets, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 121–133.

[40] S. Rei, R. Abreu, A Database of Existing Vulnerabilities to Enable Controlled Testing Studies, International Journal of Secure Software Engineering (IJSSE) 8 (3) (2017) 1–23. doi:10.4018/IJSSE.2017070101.

[41] M. Jimenez, Y. Le Traon, M. Papadakis, [Engineering Paper] Enabling the Continuous Analysis of Security Vulnerabilities with VulData7, in: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2018, pp. 56–61. doi:10.1109/SCAM.2018.00014.

[42] X. Wang, K. Sun, A. Batcheller, S. Jajodia, Detecting "0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS, in: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2019, pp. 485–492. doi:10.1109/DSN.2019.00056.

[43] J. Fan, Y. Li, S. Wang, T. N. Nguyen, A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries, in: Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 508–512. doi:10.1145/3379597.3387501.

[44] S. Reis, R. Abreu, A ground-truth dataset of real security patches (Oct. 2021). doi:10.48550/arXiv.2110.09635.
URL http://arxiv.org/abs/2110.09635

[45] G. Bhandari, A. Naseer, L. Moonen, CVEfixes: automated collection of vulnerabilities and their fixes from open-source software, in: Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2021, Association for Computing Machinery, New York, NY, USA, 2021, pp. 30–39. doi:10.1145/3475960.3475985.

[46] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, C. Dangremont, A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software (Mar. 2019). doi:10.48550/arXiv.1902.02595.
URL http://arxiv.org/abs/1902.02595

[47] Vulncode-DB, https://www.vulncode-db.com/ (2024).

[48] X. Tan, Y. Zhang, C. Mi, J. Cao, K. Sun, Y. Lin, M. Yang, Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 3282–3299. doi:10.1145/3460120.3484593.

[49] J. Zhang, X. Hu, L. Bao, X. Xia, S. Li, Dual prompt-based few-shot learning for automated vulnerability patch localization, in: 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2024, pp. 940–951. doi:10.1109/SANER60148.2024.00102.

[50] K. Shen, Y. Zhang, L. Bao, Z. Wan, Z. Li, M. Wu, Patchmatch: A tool for locating patches of open source project vulnerabilities, in: 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2023, pp. 175–179. doi:10.1109/ICSE-Companion58688.2023.00049.

[51] Z. Han, X. Li, Z. Xing, H. Liu, Z. Feng, Learning to predict severity of software vulnerability using only vulnerability description, in: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 125–136. doi:10.1109/ICSME.2017.52.

[52] N. Marastoni, R. Giacobazzi, M. Dalla Preda, A deep learning approach to program similarity, in: Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis, MASES 2018, Association for Computing Machinery, New York, NY, USA, 2018, pp. 26–35. doi:10.1145/3243127.3243131.

[53] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, M. McConley, Automated vulnerability detection in source code using deep representation learning, in: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), 2018, pp. 757–762. doi:10.1109/ICMLA.2018.00120.

[54] Y. Mao, Y. Li, J. Sun, Y. Chen, Explainable software vulnerability detection based on attention-based bidirectional recurrent neural networks, in: 2020 IEEE International Conference on Big Data (Big Data), 2020, pp. 4651–4656. doi:10.1109/BigData50022.2020.9377803.

[55] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, Vuldeepecker: A deep learning-based system for vulnerability detection, in: Proceedings 2018 Network and Distributed System Security Symposium, NDSS 2018, Internet Society, 2018. doi:10.14722/ndss.2018.23158.

[56] H. Wei, M. Li, Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code, in: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17, 2017, pp. 3034–3040. doi:10.24963/ijcai.2017/423.

[57] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, Sysevr: A framework for using deep learning to detect software vulnerabilities, IEEE Transactions on Dependable and Secure Computing 19 (4) (2022) 2244–2258. doi:10.1109/tdsc.2021.3051525.

[58] Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu, Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, in: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, R. Garnett (Eds.), Advances in Neural Information Processing Systems, Vol. 32, Curran Associates, Inc., 2019.

[59] X. Cheng, H. Wang, J. Hua, G. Xu, Y. Sui, Deepwukong: Statically detecting software vulnerabilities using deep graph neural network, ACM Trans. Softw. Eng. Methodol. 30 (3) (Apr. 2021). doi:10.1145/3436877.

[60] S. Cao, X. Sun, L. Bo, Y. Wei, B. Li, Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection, Information and Software Technology 136 (2021) 106576. doi:https://doi.org/10.1016/j.infsof.2021.106576.

[61] D. Hin, A. Kan, H. Chen, M. A. Babar, Linevd: statement-level vulnerability detection using graph neural networks, in: Proceedings of the 19th International Conference on Mining Software Repositories, MSR '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 596–607. doi:10.1145/3524842.3527949.

[62] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, S. Nepal, Transformer-based language models for software vulnerability detection, in: Proceedings of the 38th Annual Computer Security Applications Conference, ACSAC '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 481–496. doi:10.1145/3564625.3567985.

[63] C. Arora, J. Grundy, M. Abdelrazek, Advancing requirements engineering through generative ai: Assessing the role of llms, in: Generative AI for Effective Software Development, Springer, 2024. doi:10.1007/978-3-031-55642-5_6.

[64] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, Q. Wang, ClarifyGPT: A framework for enhancing LLM-based code generation via requirements clarification, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE), 2024, to appear in ESEC/FSE-2024 Research Track.

[65] C. S. Xia, Y. Wei, L. Zhang, Automated program repair in the era of large pre-trained language models, in: Proceedings of the 45th International Conference on Software Engineering (ICSE), IEEE Press, 2023, pp. 1482–1494.

[66] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, Y. Lou, No more manual tests? evaluating and improving chatgpt for unit test generation, Proceedings of the ACM on Software Engineering (ESEC/FSE) 1 (Issue FSE) (2024) Article 76, 24 pages. doi:10.1145/3624032.3624035.

[67] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W. Chiang, Y. Lyu, H. Nguyen, O. Tripp, A deep dive into large language models for automated bug localization and repair, Proceedings of the ACM on Software Engineering (ESEC/FSE) 1 (Issue FSE) (2024) 1471–1493. doi:10.48550/arXiv.2404.11595.

[68] C. Zhang, H. Liu, J. Zeng, K. Yang, Y. Li, H. Li, Prompt-enhanced software vulnerability detection using chatgpt, in: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE-Companion '24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 276–277. doi:10.1145/3639478.3643065.

[69] J. Bae, S. Kwon, S. Myeong, Enhancing software code vulnerability detection using gpt-4o and claude-3.5 sonnet: A study on prompt engineering techniques, Electronics 13 (13) (2024). `doi:10.3390/electronics13132657`.

[70] Y. Nong, M. Aldeen, L. Cheng, H. Hu, F. Chen, H. Cai, Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities (2024). `arXiv:2402.17230`.
URL `https://arxiv.org/abs/2402.17230`

[71] J. Bae, S. Kwon, S. Myeong, Enhancing software code vulnerability detection using gpt-4o and claude-3.5 sonnet: A study on prompt engineering techniques, Electronics 13 (13) (2024). `doi:10.3390/electronics13132657`.

[72] H. Pearce, B. Tan, B. Ahmad, R. Karri, B. Dolan-Gavitt, Examining zero-shot vulnerability repair with large language models, in: 2023 IEEE Symposium on Security and Privacy (SP), 2023, pp. 2339–2356. `doi:10.1109/SP46215.2023.10179324`.

[73] Y. Nong, H. Yang, L. Cheng, H. Hu, H. Cai, Automated software vulnerability patching using large language models (2024). `arXiv:2408.13597`.
URL `https://arxiv.org/abs/2408.13597`

[74] U. Kulsum, H. Zhu, B. Xu, M. d'Amorim, A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback (2024). `arXiv:2405.15690`.
URL `https://arxiv.org/abs/2405.15690`