VulAdvisor: Natural Language Suggestion Generation for Software Vulnerability Repair

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

Chong Wang*
Nanyang Technological University
Singapore
chong.wang@ntu.edu.sg

Anran Li
Yale University
USA
anran.li@yale.edu

Wenhan Wang
University of Alberta
Canada
wenhan12@ualberta.ca

Tianlin Li
Nanyang Technological University
Singapore
tianlin001@e.ntu.edu.sg

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

ABSTRACT

Software vulnerabilities pose serious threats to the security of modern software systems. Deep Learning-based Automated Vulnerability Repair (AVR) has gained attention as a potential solution to accelerate the remediation of vulnerabilities. However, recent studies indicate that existing AVR approaches often only generate patches, which may not align with developers' current repair practices or expectations. In this paper, we introduce VulAdvisor, an automated approach that generates natural language suggestions to guide developers or AVR tools in repairing vulnerabilities. VulAdvisor comprises two main components: oracle extraction and suggestion learning. To address the challenge of limited historical data, we propose an oracle extraction method facilitating ChatGPT to construct a comprehensive and high-quality dataset. For suggestion learning, we take the supervised fine-tuning CodeT5 model as the basis, integrating local context into Multi-Head Attention and introducing a repair action loss, to improve the relevance and meaningfulness of the generated suggestions. Extensive experiments on a large-scale dataset from real-world C/C++ projects demonstrate the effectiveness of VulAdvisor, surpassing several alternatives in terms of both lexical and semantic metrics. Moreover, we show that the generated suggestions enhance the patch generation capabilities of existing AVR tools. Human evaluations further validate the quality and utility of VulAdvisor's suggestions, confirming their potential to improve software vulnerability repair practices.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Software testing and debugging*.

KEYWORDS

vulnerability repair, large language models, suggestion generation, program repair

*Corresponding author.

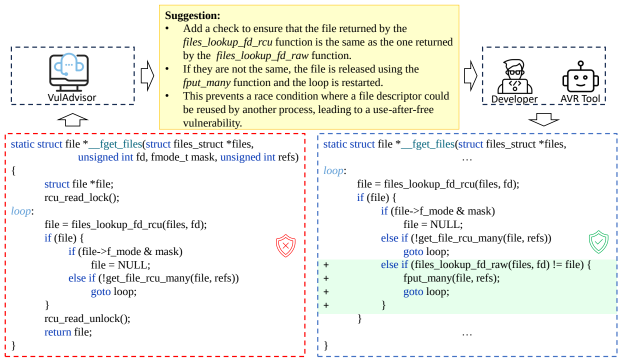
ACM Reference Format:
Jian Zhang, Chong Wang, Anran Li, Wenhan Wang, Tianlin Li, and Yang Liu. 2024. VulAdvisor: Natural Language Suggestion Generation for Software Vulnerability Repair. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695555

1 INTRODUCTION

Software vulnerabilities are a type of weakness or flaw that can be introduced at any stage of the software life cycle, such as during the design, coding, and maintenance phases, making it exploitable by attackers [24, 43]. When exploited, vulnerabilities can lead to a negative impact on the confidentiality, integrity, or availability of data and systems [1]. Despite considerable efforts to enhance software security, new vulnerabilities continually emerge in modern software. For example, the number of exposed vulnerabilities recorded by the National Vulnerability Database (NVD) [3] grows rapidly and already exceeds two hundred thousand, let alone the ones that have not been publicly disclosed or remain unknown. Such a fact highlights the urgent need of automatic techniques for accelerating the remediation of vulnerabilities, as attackers may already be aware of them and exploiting them for malicious purposes (e.g., the Heartbleed attack [2]).

To this end, some recent studies propose DL-based approaches for automated vulnerability repair by learning from the vulnerability-fix pairs, based on traditional neural models or large language models (LLMs) [6, 12, 14, 33, 69]. For example, Chen et al. [6] introduced VRepair based on transfer learning to deal with the problem of insufficient vulnerability data. VRepair first trained a Transformer model on a large bug fix corpus and then transferred it to a vulnerability fix dataset for patch generation. As a representative of LLM-based approach, VulRepair [12] fine-tunes a pre-trained model CodeT5 on real-world vulnerability fixes to generate patches for given vulnerable functions, and shows promising results for vulnerability fixing at a large scale.

However, existing vulnerability repair approaches primarily focus on generating patches, which may not fully meet the needs of developers. First, developers worry about incurring high maintenance costs, particularly if the generated patches are often unclear or incorrect [36, 47]. Unfortunately, even with the use of perfect



Suggestion:
- Add a check to ensure that the file returned by the *files_lookup_fd_rcu* function is the same as the one returned by the *files_lookup_fd_raw* function.
- If they are not the same, the file is released using the *fput_many* function and the loop is restarted.
- This prevents a race condition where a file descriptor could be reused by another process, leading to a use-after-free vulnerability.

```
static struct file *__fget_files(struct files_struct *files,
                unsigned int fd, fmode_t mask, unsigned int refs)
{
        struct file *file;
        rcu_read_lock();
loop:
        file = files_lookup_fd_rcu(files, fd);
        if (file) {
                if (file->f_mode & mask)
                        file = NULL;
                else if (!get_file_rcu_many(file, refs))
                        goto loop;
        }
        rcu_read_unlock();
        return file;
}
```

```
static struct file *__fget_files(struct files_struct *files,
                                 …
loop:
        file = files_lookup_fd_rcu(files, fd);
        if (file) {
                if (file->f_mode & mask)
                        file = NULL;
                else if (!get_file_rcu_many(file, refs))
                        goto loop;
+               else if (files_lookup_fd_raw(files, fd) != file) {
+                       fput_many(file, refs);
+                       goto loop;
+               }
        }
                                 …
}
```

Figure 1: An illustrative example of suggestion generation for vulnerability repair from CVE-2021-4083.

localization, the performance of current AVR tools remains suboptimal, with success rates often below 20% [12, 69]. Second, recent studies [5, 55] indicate a strong preference among developers to remain involved in the repair process by manually crafting patches, rather than relying on fully automated solutions. This preference is mainly due to trust issues with automatically generated patches. Therefore, it is crucial not only to improve the quality of vulnerability patches but also to provide supplementary information that aids developers in the vulnerability fixing process.

In this paper, we propose to facilitate developers repair software vulnerability by automatically generating suggestions for it, namely VulAdvisor. Instead of solely providing subjects with automatically generated patches, our approach is developer-centric that provides suggestions in the form of natural language. As illustrated in Figure 1, suppose there is a function *__fget_files* reported to be vulnerable by vulnerability detection tools or users, the developer should address the vulnerability from scratch, as no information is provided regarding where or how to implement the patch. As mentioned previously, at this stage, it is beneficial to suggest the developer the steps to fix the vulnerability and the potential reason behind it. When given the generated suggestion, it is clearer that one should check the consistency of returned files and the raw file by *files_lookup_fd_rcu* and *files_lookup_fd_raw* function calls respectively. Furthermore, we can explain the way the provided patches related to the vulnerability, that is, to prevent a race condition. As such, the developer can successfully repair the vulnerability with low manual effort and maintenance cost. Inspired by the success of Large Language Models (LLMs), we design VulAdvisor, which consists of two main components: Oracle Extraction and Suggestion Learning. These components are based on ChatGPT [38] and CodeT5 [53], respectively.

Oracle Extraction. One fundamental challenge in automating suggestion generation is the scarcity of sufficient historical data. A straightforward approach to obtain oracles (i.e., ground truths) is to simply prompt ChatGPT with, "How to fix the vulnerability in the following function? <code>". However, devising repair steps for vulnerabilities requires deep domain knowledge, a challenge even for experienced security experts [51]. Therefore, expecting ChatGPT to independently generate viable suggestions is impractical. To address this, we gather historical patches from CVE/NVD and prompt ChatGPT to summarize how the vulnerability was fixed based on its patch commit. This task is much easier for ChatGPT due to its human-like proficiency in code and text summarization [31, 48]. Afterwards, we revert the commit to obtain its corresponding vulnerable function and post-process the summary into a suggestion. This approach allows us to build a comprehensive dataset with high-quality samples by extracting pairs of vulnerable functions and their repair suggestions.

Suggestion Learning. We formulate the problem of suggestion generation as a sequence-to-sequence problem. We introduce a neural model based on CodeT5, which has demonstrated its effectiveness in vulnerability repair [12, 69]. By fine-tuning CodeT5 on our curated dataset, the model encodes the vulnerable function and employs its decoder to generate suggestions. Nevertheless, there are still two aspects for improvement. On the side of vulnerable function, LLMs can sometimes neglect local context, leading to suggestions that include irrelevant information (e.g., nonexistent variables) [35, 44]. Therefore, we design a localness-based attention mechanism inspired by the localness of source code [50]. We apply the Vector Space Model (VSM) to identify locally specific terms (such as variables) within the function and compute their TF-IDF weights. These weights are then used to increase their awareness in Multi-Head Attention, encouraging it to adhere to the local context of the code. On the other side of suggestion, the words can have different impacts on the repair behavior. Intuitively, repair actions comprising predicates and their objects represent the backbone of the suggestion, making them much more meaningful than common words. To highlight their importance while maintaining fluency, we design a repair action loss that additionally considers the loss of repair action words during fine-tuning. Notably, these two strategies are only used for guiding the learning of the base LLM. During inference, the fine-tuned model can be directly used for generating suggestions without these strategies, making it a reusable foundation for further improvement.

We build a large-scale dataset by collecting pairs of vulnerability and its suggestion with our oracle extraction method from real-world C/C++ projects. Our initial step involved a manual inspection of this dataset, revealing that approximately 96.9% of the ground-truth suggestions are of high quality. This encouraging finding prompted us to undertake extensive experiments to thoroughly evaluate our approach. We consider a variety of popular text generation approaches as baselines. In addition to the lexical and semantic metrics such as BLEU, ROUGE, and BERTScore, we propose a Repair Action Score (RAS) for evaluation. The experimental results demonstrate that VulAdvisor significantly outperforms the baselines across all metrics. Furthermore, we explored the usefulness of the generated suggestions by integrating them into VulRepair, observing a notable enhancement (e.g., 5.1% of EM) in patch generation. We also conduct a human evaluation to assess the quality and practical utility of generated suggestions, and the results further confirm the effectiveness of VulAdvisor. In summary, the main contributions of this work are as follows:

- We formulate a new problem of automated suggestion generation for vulnerability repair and propose a novel method for building the dataset. To the best of our knowledge, this is the first work that aims to generate natural language suggestions

for fixing the vulnerabilities, which can help the process of vulnerability repair from a developer's perspective.
- We propose a novel LLM-based approach for generating vulnerability repair suggestions. The approach consists of a localness-based attention mechanism that augments the CodeT5 model with additional local context for relevance, and a repair action loss that focuses on learning meaningful words.
- We provide a large-scale dataset consisting of over 18k of vulnerability-suggestion pairs from real-world C/C++ projects and conduct extensive experiments on it. The evaluation demonstrates the effectiveness of our approach in terms of four automatic metrics and human evaluation results.

2 PROBLEM AND PRELIMINARY

2.1 Problem Definition

Our objective is to automatically generate suggestions that provide guidance for fixing newly disclosed vulnerabilities. Additionally, we aim to facilitate their understanding of the solutions, particularly when they may lack expertise in security matters. Therefore, we formulate the problem of *vulnerability repair suggestion*, which involves recommending practical solutions for software vulnerabilities from a developer's perspective. We formally define it as follows.

Vulnerability Repair Suggestion. Let $C = (c_1, c_2, \ldots, c_K)$ demotes the source code of a vulnerable function, where $c_i, i \in [1, K]$ are the tokens in the function. The input of this task is the sequence of tokens c_1 to c_K containing vulnerable elements. The target is to learn a generator $\psi : \mathcal{C} \rightarrow \mathcal{S}$ that generates the suggestion $S = (s_1, s_2, \ldots, s_L)$ in the mixed form of natural language and code elements. Here $\mathcal{C}$ and $\mathcal{S}$ represent all vulnerable functions and suggestions respectively. The generated suggestion should be able to guide the repair of the vulnerability in c, and explain the possible reason behind the modification if necessary. Here s_i is the i-th token in the generated sequence of length L.

2.2 Use Scenarios

2.2.1 Providing Developer Guidance. The problem is grounded on the hypothesis that the vulnerable function has been identified either by a static detector or reported by the user, but with limited detailed information available. Consequently, fixing the vulnerability often requires cumbersome manual effort. By providing targeted suggestions for the detected vulnerability, developers can not only follow the steps to successfully repair it, but also gain insight into the fix and easily verify the soundness of the suggestions. Moreover, our defined problem is readily applicable to other scenarios. For instance, when additional clues such as error messages or stack traces at the trigger point are available, we can incorporate them as joint inputs to generate more refined suggestions.

2.2.2 Improving DL-based AVR Tools. Existing DL-based Automated Vulnerability Repair (AVR) tools typically assume perfect fault localization, where the vulnerable function and all locations for modification are given. While effective, this assumption does not align with the reality faced by developers. In fact, when provided with only the function to be fixed, the performance of these tools can drop significantly [30, 37]. Also, the complexity of vulnerability repair makes it challenging to learn accurate patches due to the lack of clear guidance on repair direction. To help bridge this gap, we can leverage generated suggestions that specifically identify faulty elements and describe how they can be fixed. We believe that such information is valuable for LLMs, guiding them to enhance the quality of patches. The effectiveness of this approach will be evaluated in our study.

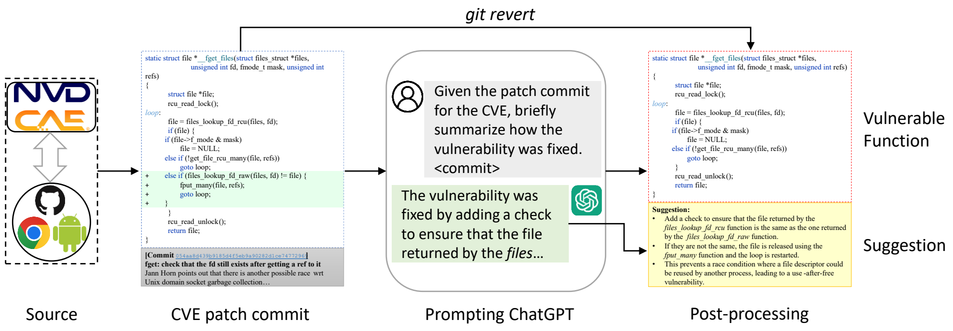
3 APPROACH

3.1 Overview

As mentioned earlier, we aim to create a synergy between automated methods and developer insights, ultimately leading to more effective and reliable vulnerability patching solutions. To this end, our approach is developer-centric that generates easy-to-understand suggestions for vulnerability repair (we name it VulAdvisor). The architecture of VulAdvisor consists of two main modules: *oracle extraction* and *suggestion learning*. On the one hand, since there is no existing dataset available for training neural networks, we propose an oracle extraction method to prepare the vulnerable functions and the corresponding suggestions based on the vulnerability patch commits and ChatGPT respectively. On the other hand, when we obtain sufficient pairs of data described above, we adopt a popular encoder-decoder LLM of code (i.e., CodeT5) to automatically learn the semantics of vulnerability and generate the suggestions in the mixed form of natural language and code elements. In addition to fine-tuning CodeT5, we enhance the model by incorporating localness context and employ a repair action loss to improve the quality of the generated suggestions.

3.2 Oracle Extraction

In general, although LLMs of code are pre-trained and capable of learning fundamental knowledge related to software bugs and vulnerabilities, their performance in program repair tasks greatly benefits from high-quality data for fine-tuning and adaptation [12, 19]. However, different from patch generation where the historical vulnerability fixes can be easily collected, to date, there still lacks of existing data specific for suggestion generation. Initially, our approach involved merely using the commit message of a patch as the basis for generating suggestions, assuming that it would fully describe the changes made to fix the vulnerability. While in practice, previous studies have revealed that the quality of commit messages varies due to a lack of motivation or time, for example, nearly 66% of the messages are not informative and contained only a few words [9, 49]. Given the remarkable capabilities demonstrated by ChatGPT, a promising solution is to directly prompt ChatGPT by asking it how to fix the vulnerability. Unfortunately, the truth is that fixing vulnerabilities requires substantial domain knowledge, far beyond simply understanding the intention of the code. Consequently, ChatGPT cannot complete such a highly challenging task. Upon analyzing some of the results, we observed that ChatGPT tended to propose multiple best practices for security development, even if they were not directly relevant to fixing the specific vulnerability (more details will be presented in Section 4).



```
static struct file *__fget_files(struct files_struct *files,
                    unsigned int fd, fmode_t mask, unsigned int
refs)
{
          struct file *file;
          rcu_read_lock();
loop:
          file = files_lookup_fd_rcu(files, fd);
          if (file) {
        if (file->f_mode & mask)
               file = NULL;
        else if (!get_file_rcu_many(file, refs))
               goto loop;
+       else if (files_lookup_fd_raw(files, fd) != file) {
+              fput_many(file, refs);
+              goto loop;
+       }
          }
          rcu_read_unlock();
          return file;
}
```

[Commit 054aa8d439b9185d4f5eb9a90282d1ce74772969]
fget: check that the fd still exists after getting a ref to it
Jann Horn points out that there is another possible race wrt Unix domain socket garbage collection…

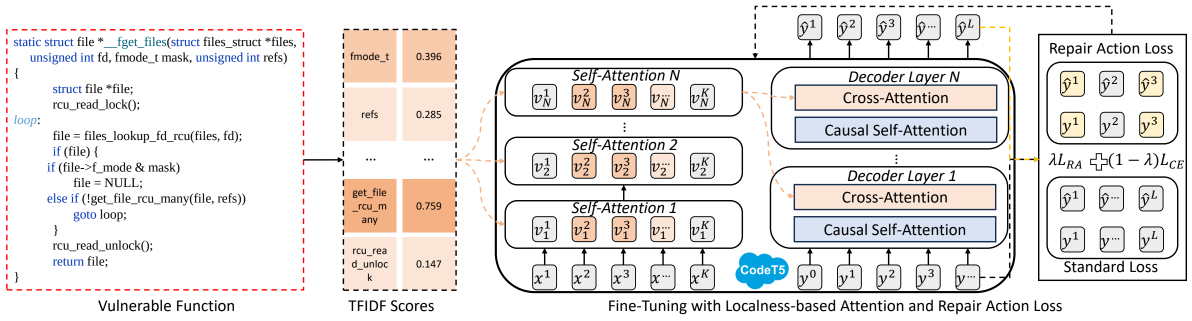
Given the patch commit for the CVE, briefly summarize how the vulnerability was fixed. <commit>

The vulnerability was fixed by adding a check to ensure that the file returned by the *files*…

```
static struct file *__fget_files(struct files_struct *files,
                    unsigned int fd, fmode_t mask, unsigned int refs)
{
          struct file *file;
          rcu_read_lock();
loop:
          file = files_lookup_fd_rcu(files, fd);
          if (file) {
        if (file->f_mode & mask)
               file = NULL;
        else if (!get_file_rcu_many(file, refs))
               goto loop;
          }
          rcu_read_unlock();
          return file;
}
```

Suggestion:
- Add a check to ensure that the file returned by the *files_lookup_fd_rcu* function is the same as the one returned by the *files_lookup_fd_raw* function.
- If they are not the same, the file is released using the *fput_many* function and the loop is restarted.
- This prevents a race condition where a file descriptor could be reused by another process, leading to a use -after-free vulnerability.

Figure 2: The workflow of oracle extraction.


fmode_t
0.396
refs
0.285
…
…
get_file_rcu_many
0.759
rcu_read_unlock
0.147
Self-Attention N
Self-Attention 2
Self-Attention 1
v_N^1 v_N^2 v_N^3 $v_N^{\cdots}$ v_N^K
v_2^1 v_2^2 v_2^3 $v_2^{\cdots}$ v_2^K
v_1^1 v_1^2 v_1^3 $v_1^{\cdots}$ v_1^K
x^1 x^2 x^3 $x^{\cdots}$ x^K
CodeT5
Decoder Layer N
Decoder Layer 1
Cross-Attention
Causal Self-Attention
y^0 y^1 y^2 y^3 $y^{\cdots}$
$\hat{y}^1$ $\hat{y}^2$ $\hat{y}^3$ $\hat{y}^{\cdots}$ $\hat{y}^L$
Repair Action Loss
$\hat{y}^1$ $\hat{y}^2$ $\hat{y}^3$
y^1 y^2 y^3
$\lambda L_{RA} + (1-\lambda) L_{CE}$
$\hat{y}^1$ $\hat{y}^{\cdots}$ $\hat{y}^L$
y^1 $y^{\cdots}$ y^L
Standard Loss
Vulnerable Function
TFIDF Scores
Fine-Tuning with Localness-based Attention and Repair Action Loss

```
static struct file *__fget_files(struct files_struct *files,
    unsigned int fd, fmode_t mask, unsigned int refs)
{
        struct file *file;
        rcu_read_lock();
loop:
        file = files_lookup_fd_rcu(files, fd);
        if (file) {
      if (file->f_mode & mask)
            file = NULL;
      else if (!get_file_rcu_many(file, refs))
            goto loop;
        }
        rcu_read_unlock();
        return file;
}
```

Figure 3: The framework of our suggestion learning approach.

In order to facilitate ChatGPT in providing useful suggestions as an oracle[1], we transform the problem into a summarization task. Figure 2 illustrates the process of oracle extraction based on ChatGPT. Specifically, we first collect a large number of vulnerability patch commits from open-source repositories according to CVE/NVD records. For simplicity, we use *commit* to represent them. Instead of expecting ChatGPT to come up with a solution from scratch, a specific and effective prompt is designed: *"Given the patch commit for the CVE, briefly summarize how the vulnerability was fixed. <commit>"*. The rationale behind this prompt is that developers often can easily understand how a vulnerability was fixed according to the code changes and commit message [18], similar to the concept of the "Monday morning quarterback". In this way, we enable ChatGPT to focus on describing the changes and providing reasoning for the fixes. By transforming the problem into a summarization task and utilizing the *commit* information as input, ChatGPT is encouraged to concentrate solely on the changes made to address the vulnerability. As a result, it can generate well-crafted summaries that describe the essence of the vulnerability fixes.

After performing fix summarization, we revert the patched version to the previous one, and extract the vulnerable function from it. Then we further refine the summary into a suggestion using post-processing. The summary typically (in 99.4% of cases) starts with the phrase "The vulnerability was fixed by", which is not informative and has a negative impact on the evaluation. Hence, we remove the introductory phrase and lemmatize the verb following it. Additionally, we replace the past tense with the present tense to enhance the readability of the suggestion. Through these adjustments, we have curated a dataset suitable for training models to generate vulnerability fix suggestions.

3.3 Suggestion Learning

Although various LLMs exist for code-related tasks, we align with recent works in vulnerability repair by choosing CodeT5 [12, 52]. This decision is based on CodeT5's advantageous balance of effectiveness and efficiency in this domain. We can fine-tune it on our dataset by encoding the vulnerable function and decoding it to generate the suggestion. Though effective to some extent, such a practice still has two limitations. First, from the NLP community, there is evidence that LLMs are prone to overlook context [35, 44], and thus yield irrelevant content. We expect that the generated text should match with the original relations and facts in the local context, for example, the variables suggested for repair. It goes beyond mere accuracy of a model's performance, but concerns the validity and reliability of its suggestion process, which is crucial for building a developer-centric tool [23, 56]. Second, the standard cross-entropy loss in the fine-tuning phase assigns equal training weights to all target tokens without considering their semantic importance. This approach can undervalue words that significantly contribute to the overall semantic correctness. Intuitively, words that predominantly determine the repair behavior of a vulnerability (e.g., "Add check") carry crucial semantic information. However, these important tokens may be underrepresented when minimizing the loss, resulting in generated suggestions that may lack meaningfulness and effectiveness in addressing the vulnerability.

[1]In this paper, we use 'oracle' and 'ground-truth' interchangeably.

For localness-based attention, we begin by computing TF-IDF scores of terms using a Vector Space Model (VSM). We then map these scores to the subtokens of the input and increase the weights in both the Self-Attention and Cross-Attention mechanisms. This enhancement aims to improve the model's focus on local context. In terms of the repair action loss, we identify repair action words in the suggestion and map them into subwords. We then calculate a weighted sum with the original sentence and these subwords to compute the loss for updating the model. This approach helps the model learn from repair actions, improving its ability to suggest effective repairs for vulnerable functions.

Now we describe the process of supervised fine-tuning CodeT5 with the two strategies for the task of vulnerability repair suggestion. CodeT5 uses an encoder-decoder Transformer framework, where the encoder processes the vulnerable function, and the decoder generates the suggestion. The encoder/decoder stack consists of multiple pre-trained attentional layers, each refining the contextual information by attending to different parts of the input. For illustration purposes, we denote the supervised dataset as $D = \{(c^i, s^i)\}_{i=1}^{N}$ of size N examples.

3.3.1 Localness-Based Attention. The localness of source code is originally introduced by Tu et al. [50], where human-written programs are localized and have useful local regularities (e.g., endemic and specific n-grams) that can be captured and exploited. Therefore, emphasizing such local context may contribute to the improvement of LLMs for generating more faithful tokens, since they tend to ignore the context as discussed above. Similarly, researchers also found that programmers usually focus on a list of keywords when reading and understanding code [41]. Inspired by these studies, we propose to enrich the localness of a vulnerable function through the VSM to select keywords/terms from it.

We use the Term Frequency-Inverse Document Frequency (TF-IDF) algorithm to achieve the goal. Specifically, we treat functions as documents and keyword tokens (e.g., variable names, function calls) as terms. TF-IDF gives higher importance to terms that appear frequently within a function but are relatively rare across the entire corpus (i.e., all functions). This helps to identify keywords in locality. Formally, given a vulnerable function $c = (c_1, c_2, \ldots, c_K)$, TFIDF weight for term c_i in the function c, denoted as $\mathrm{tfidf}(c_i, c)$, can be computed using the following formula:

$$\mathrm{tfidf}(c_i, c) = \frac{\mathrm{count}(c_i, c)}{\sum_{j=1}^{K} \mathrm{count}(c_j, c)} \times \log\left(\frac{N}{\mathrm{df}(c_i)}\right) \tag{1}$$

where $\mathrm{count}(c_i, c)$ is the number of occurrences of token c_i in function c, N is the total number of functions in the corpus, and $\mathrm{df}(c_i)$ is the number of functions containing token c_i.

When we obtain the TF-IDF weights for the tokens in C, we map them to the subtokens from BPE of CodeT5. For example, in Figure 3, the term "fmode_t" will be split into subtokens of "f", "mode", and "_t", all of which will have the same score of 0.396. We denote the subtoken sequence of c as x, and let $M_x = (M_{x,1}, M_{x,2}, \ldots, M_{x,C'})$ be the TF-IDF weights for these subtokens. To incorporate the localness-based attention mechanism into our model, we modify the multi-head attention mechanism in CodeT5. Let $\mathbf{Q_e}$, $\mathbf{K_e}$, and $\mathbf{V_e}$ be the query, key, and value matrices, respectively, for the original multi-head attention. For each encoder stack, we compute the updated attention value matrices $\mathbf{V'_e}$ as follows:

$$\mathbf{V'_e} = \mathrm{softmax}\left(\frac{\mathbf{Q_e}\mathbf{K_e}^{\top}}{\sqrt{d_k}} + \alpha M_x\right)\mathbf{V_e} \tag{2}$$

where d_k is the dimension of the key vectors, and M_x is a row vector containing the TF-IDF weights for each subtoken in x. The factor α is introduced to control the alignment of the original attention scores. The softmax operation is applied along the subtoken dimension to ensure that the attention weights sum to 1 for each head.

Typically, the pretrained encoder has T stacks, and we denote the output of the final stack as $\mathbf{V'}_{e,T}$. Similarly, the enhanced attention weights of cross-attention $\mathbf{V'_d}$ of the decoder are computed as:

$$\mathbf{V'_d} = \mathrm{softmax}\left(\frac{\mathbf{Q_d}\mathbf{K_d}^{\top}}{\sqrt{d_k}} + \alpha M_x\right)\mathbf{V'}e, T \tag{3}$$

By incorporating the localness-based attention mechanism using TF-IDF weights, the model is encouraged to focus more on the locally specific terms within the vulnerable function, thus improving the hidden states $\mathbf{V'}_d$ and its ability to generate repair suggestions that adhere to the local context of the code.

3.3.2 Repair Action Loss. When optimizing the model, instead of using the standard cross-entropy loss, we propose a repair action loss to cater to the repair actions for improving the meaningfulness of generated suggestions. We define a Repair Action (RA) as a specific operation or modification in a suggestion that represent the minimal unit of change to address a vulnerability of the code. Repair actions typically consist of verbs (actions) and their corresponding objects (entities affected by the actions), such as variable names, function calls, or specific code snippets.

Algorithm 1 Identifying Repair Actions in Suggestions

```
Require: word_ids: The map for the original words and subwords
    from BPE.
Ensure: ra_mask: Mask indicating positions of repair actions.
 1: doc ← POS of the suggestion
 2: for all token ∈ doc do
 3:     if token.pos_ = VERB then
 4:         verb_idx ← token.i
 5:         for all child ∈ token.children do
 6:             if child.dep_ ∈ {dobj, iobj, pobj, nsubjpass} then
 7:                 obj_idx ← child.i
 8:                 verb_tokens ← []
 9:                 obj_tokens ← []
10:                 for all idx, word_id ∈ enumerate(word_ids) do
11:                     if word_id = verb_idx then
12:                         verb_tokens.append(idx)
13:                     else if word_id = obj_idx then
14:                         obj_tokens.append(idx)
15:                 for all vt ∈ verb_tokens do
16:                     ra_mask[vt] ← 1
17:                 for all ot ∈ obj_tokens do
18:                     ra_mask[ot] ← 1
19: return ra_mask
```

Algorithm 1 outlines the process of identifying repair actions and creating their masks from given suggestions. The algorithm begins by analyzing the part-of-speech (POS) of each token in the input suggestion. For tokens identified as verbs, the algorithm examines their children to find the object of the verb. In Line 6, *dobj* refers to the direct object, *iobj* to the indirect object, and *pobj* to the prepositional object. We also consider *nsubjpass* (passive nominal subject), which represents the noun or noun phrase that acts as the subject in a passive voice construction. It then maps the indices of these verb and object tokens to their corresponding positions in the Byte Pair Encoding (BPE) subwords, using a word IDs mapping (Lines 10-14). This process creates a mask ('ra_mask') that highlights the positions of these identified repair actions in the suggestion (Lines 15-18).

Next, we incorporate this repair action mask into the loss function to prioritize the correct generation of these important elements. The repair action loss $\mathcal{L}_{\text{RA}}$ is computed as:

$$\mathcal{L}_{\text{RA}} = -\sum_{i=1}^{L} \text{ra\_mask}_i \log P(y_i | y_{<i}) \tag{4}$$

where L is the length of the generated sequence, ra_mask_i is the repair action mask value for the i-th token, and $P(y_i)$ is the probability of the i-th token in the generated sequence and is calculated based on the contextual representation of $\mathbf{V'_d}$.

The final loss function is a weighted sum of the repair action loss ($\mathcal{L}_{\text{RA}}$) and the standard cross-entropy loss ($\mathcal{L}_{\text{CE}}$). The combined loss is calculated as:

$$\mathcal{L}_{\text{final}} = \lambda \cdot \mathcal{L}_{\text{RA}} + (1 - \lambda) \cdot \mathcal{L}_{\text{CE}}$$

where λ is a hyperparameter that controls the balance between the two losses. By incorporating the repair action loss, our model is incentivized to optimize the words that can better represent the suggestion for repairing the vulnerability, thereby improving the general quality.

4 EVALUATION

4.1 Dataset

4.1.1 Dataset Construction. To automate the task of vulnerability repair suggestion and evaluate our approach, we curated a large-scale dataset consisting of pairs of vulnerable functions and corresponding suggestions based on CVEs and open-source projects. To begin, we crawled all the CVEs up to the year 2023 and selected those with a patch containing C/C++ files. We parsed the file before the fixed version to extract the vulnerable functions and extracted the fixed functions from the patch files. Using "git diff," we analyzed the code changes between the two functions, referred to as *diff* s. We excluded instances where the vulnerable function could not be found in the fixed version to maintain dataset quality. In total, we obtained 35,699 *diff* s.

To further enhance the data quality, we filtered out duplicated or very short vulnerable functions (i.e., less than 5 lines), resulting in 20,867 *diff* s remaining. The short functions include those consist of one or two single statements or a single-line control structure within the function body. These functions typically span 3 or 4 lines and are generally easy for developers to understand. Consequently, developers may not use a tool for repair suggestions in these straightforward cases. Next, we prepared ground-truth suggestions using our oracle extraction method outlined in Section 3.2. For this purpose, we utilized the API of ChatGPT [2] (gpt-3.5-turbo), as it already met our quality needs. Nevertheless, this approach leaves room for future exploration and potential improvements using GPT-4. We ignored those responses that indicate an error (e.g., Timeout). Then we handled and discarded those only summarize line comment changes to clean the returned summaries as suggestions, and paired them with the corresponding vulnerable functions.

As a result, we constructed a dataset comprising 18,517 pairs of vulnerable functions and suggestions. On average, both sides of the pairs contain 267 and 55 tokens, respectively. The total number of sub-word tokens for the prompt and completion is 21,155,861, incurring a charge of approximately 35.96$. It is worth noting that our dataset not only has a larger volume of vulnerability samples compared to existing resources like BigVul [10], but it also represents the first artifact of its kind, providing a valuable resource for the automation of vulnerability fix suggestions.

4.1.2 Quality Assessment. One key concern for us was the quality of the constructed dataset. To investigate this, we conducted a statistically representative random sample of 384 data points, achieving a 95% confidence level with a 10% confidence interval. Subsequently, two of the authors manually inspected each sample and categorized them as 'poor,' 'fair,' 'good,' or 'excellent' based on the informativeness and naturalness. The assessment criteria are threefold, evaluating whether the suggestion accurately: 1) identifies the vulnerable code elements, 2) offers practical repair steps, and 3) explains the rationale behind the solution if necessary. When a suggestion fulfills all three criteria, we classify it as excellent, otherwise it is downgraded accordingly. In cases of disagreement, a third author participated in the discussion to reach an agreement. As a result, we identified 372 samples (96.9%) with good (40) or excellent (332) quality and 12 samples with fair quality. None of the samples were classified as poor quality. This outcome has significantly bolstered our confidence in the training and evaluation of our approach based on the prepared dataset.

4.2 Experimental Settings

We partitioned the dataset randomly into training, validation, and testing sets with fractions of 80%, 10%, and 10%, respectively. To ensure efficient processing and to accommodate the majority of cases, we set the length limits of the vulnerable function and suggestion to 400 and 100 tokens, respectively. We opted not to restrict the vocabulary since CodeT5 utilizes a pre-trained BPE tokenizer with 32k sub-tokens. For the localness-based attention, we set the alignment factor α to 1 since it fits well in our experiments. We keep this factor in our approach for generalization purposes. During supervised fine-tuning of CodeT5, we employed CodeT5-base, which has approximately 220M parameters. The training process involved 30 epochs with a batch size of 32 and utilized the Adam optimizer with a learning rate of 1e-4. We determined the optimal value for λ in the final loss to be 0.1 by iteratively testing values

[2]https://platform.openai.com/docs/guides/gpt

from 0 to 1. During inference, we utilized a beam search size of 5 to generate suggestions for vulnerability fixes.

All experiments were conducted on an Ubuntu 18.04 server with 48 cores of Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz, 256GB RAM, and 4 GTX3090 GPUs with 24GB memory.

4.3 Baselines

After conducting an extensive literature review on vulnerability repair, we discovered that there is no existing work specifically dedicated to generating vulnerability suggestions. As a result, we compare our approach with both traditional neural models and pre-trained LLMs of code, namely Seq2Seq [15, 22], Transformer [4, 46], CodeBert [11], and CodeLlama (7B) [42]. These models are commonly employed for generating text from source code. To ensure a fair comparison, we used the same hyper-parameters for these models as we did for our own approach, unless explicitly mentioned otherwise. All models were trained using their full parameters, with the exception of CodeLlama. For CodeLlama, we employed the LORA technique to facilitate its operation on our server because of its large scale. Besides, since our oracle extraction process utilizes ChatGPT, we are curious to assess its performance when provided solely with the vulnerable function. For this purpose, we use GPT-3.5 and GPT-4. The prompt format is structured as follows: *"Given the vulnerable function, please briefly suggest how to fix it within 100 words: <code>"*. We restrict the response length to 100 words in order to maintain consistency with the settings of all other approaches being evaluated.

4.4 Evaluation Metrics

To assess the performance of our approach and compare it with the baselines, we first use two lexical evaluation metrics in natural language processing tasks: ROUGE-L [28] and BLEU [39]. ROUGE-L measures the longest common subsequence between the generated and oracle suggestions. BLEU compares the n-grams (i.e., BLEU-4) of the generated text with those in the reference and computes a precision-based score. We also use the BERTScore metric [67] to evaluate the quality of the generated repair suggestions. BERTScore computes the similarity between the generated suggestions and the oracle using contextual embeddings from a pre-trained BERT model.

In addition to existing metrics, we propose a Repair Action Score (RAS) that measures the accuracy of a model for generating key repair actions. It reflects how well the model captures specific repair actions that contribute to the overall quality. The RAS is calculated using the Jaccard similarity coefficient, which measures the similarity and diversity of sets of repair actions. Given a set of repair actions generated by the model RA_m and a set of ground truth repair actions RA_g, the Jaccard similarity coefficient is calculated as:

$$RAS = \frac{1}{N}\sum_{i=1}^{N}\frac{|RA_m^i \cap RA_g^i|}{|RA_m^i \cup RA_g^i|}$$

where N is the total number of instances being evaluated.

For all metrics, the higher scores indicate better alignment between the generated suggestions and the ground-truth.

Table 1: Comparison of different approaches for vulnerability repair suggestion.

Model	BLEU	ROUGE-L	BERTScore	RAS
Seq2Seq	10.9	26.9	61.8	7.1
Transformer	6.1	24.7	58.7	6.1
CodeBert	16.7	30.8	64.2	10.0
CodeLlama-7B	12.4	27.1	63.6	6.6
GPT-3.5	3.3	17.3	56.7	1.5
GPT-4	3.2	17.0	55.9	1.5
VulAdvisor	**21.0**	**34.7**	**67.7**	**12.5**

4.5 Results

We investigate the effectiveness of our proposed approach by answering the following three research questions.

4.5.1 *RQ1. How does our approach perform on generating vulnerability repair suggestions compared to the baselines?*

Before presenting the outcomes of supervised learning-based methods, from Table 1, it is obvious that ChatGPT scored quite low across all metrics. This is particularly noticeable in the BLEU and RAS, which are in the single digits. As highlighted in Figure 1, this issue is compounded by the incorrect identification of vulnerable elements and the generation of overly general suggestions, rendering the outputs from ChatGPT of limited practical value. We expected that GPT-4 would exhibit significantly improved performance. However, to our surprise, the observed performance remained very close to GPT-3.5, or even decreased. This suggests that, without additional context or information, ChatGPT faces significant challenges in accurately predicting solutions solely based on its pre-trained knowledge.

In contrast, the classic traditional model, Seq2seq, achieves a ROUGE-L score of 26.9% and a BLEU score of 10.9%, which is much higher. The effectiveness of Seq2seq comes from its capability to learn sequential relationships, where control or data dependencies are resided in. Nevertheless, Transformer, which aims to highlight dependencies, may struggle to identify the most critical ones as it calculates all mutual information without denoising. As a result, its performance is even worse than Seq2seq (e.g., 4.8% lower in terms of absolute BLEU).

Pre-trained LLMs show a significant improvement over traditional neural model. Specifically, the ROUGE-L score of CodeBert is 30.8%, which improves Seq2seq by 3.9%. Considering the architecture is the same as Transformer, such an improvement is largely attributed to the abundance of knowledge from massive source code and documents during pre-training. When predicting the vulnerability and making suggestions after fine-tuning, CodeBert can more easily differentiate the vulnerable and benign code semantics based on the pre-trained dependencies. Contrary to our expectations for CodeLlama to yield high scores, the results turn out to be quite disappointing. Its overall performance is significantly lower compared to that of the lightweight CodeBERT (e.g., 10.0% v.s. 6.6% in RAS). We conjecture that, a decoder-only model like CodeLlama is limited to capturing dependencies only in the forward direction. Also, the large scale of parameters along with comparatively small

Table 2: Results of the ablation study for the main components of VulAdvisor.

Model	BLEU	ROUGE-L	BERTScore	RAS
SFT (CodeT5)	19.5	33.4	65.9	11.3
SFT+LA	20.4	34.2	67.3	11.6
SFT+RAL	20.6	34.5	67.5	12.2
SFT+LA+RAL	**21.0**	**34.7**	**67.7**	**12.5**

dataset, may restrict it to fit well, which may explain its relatively lower efficacy in this context.

Our proposed VulAdvisor significantly outperforms all the baselines by a large margin. For example, the improvement in BLEU score is 4.3% compared with CodeBert, which is considered a significant advancement in text generation tasks [66]. Notably, the improvement in BERTScore or RAS is as pronounced as for BLEU. This is because our underlying model, CodeT5, has a pre-trained decoder, which aids in generating fix suggestions more naturally. Moreover, the enhanced localness and repair action employed in our approach contribute to learning more precise and contextually appropriate suggestions.

4.5.2 *RQ2. How effective are the main components of VulAdvisor for the quality of generated suggestions?*

In this research question, we examine the effectiveness of the two main components of VulAdvisor, namely Localness-based Attention (LA) and Repair Action Loss (RAL), in the quality of generated suggestions. We conduct ablation experiments, where we selectively include one or more of these components. The results of the ablation study are presented in Table 2. In the ablation study, when we use the notation "SFT," it means that we simply fine-tune the pre-trained CodeT5 model.

Firstly, the addition of the LA component (SFT+LA) leads to a notable increase in performance across all metrics. For example, the ROUGE-L score improves marginally from 33.4% to 34.2%. While this increase appears modest, the BERTScore that calculates semantic similarity reveals significant enhancements. Notably, key function calls have been more accurately revised. For instance, a suggestion to replace vpx_memcpy" with vpx_memalign" is particularly relevant, considering that "vpx_memcpy" is absent in the vulnerable function. This correction addresses the issue of missing local context in the fine-tuning of Code-T5, which previously led to suggestions of irrelevant code elements.

When we integrate RAL into the supervised fine-tuning step (SFT+RAL), the model's performance also improves. In particular, the RAS, for instance, rises from 11.3% to 12.2%. This can be attributed to the emphasis of the repair actions, the consistency between training and inference in word generation. It is also noteworthy that introducing the repair action loss does not sacrifice the accuracy of other words. The BERTScore also indicates an overall improvement in semantics. This is achieved by our hyperparameter λ, which strikes a good balance between the importance of other words and repair actions.

Finally, we observe that the VulAdvisor model achieves the highest scores, demonstrating the effectiveness of the integrated approach that incorporates two additional components.

Table 3: Results of vulnerability patch generation with and without repair suggestions.

Model	BLEU	EM	Model	BLEU	EM
VRepair	36.7	7.0	VulRepair	39.6	9.9
VRepair+GT	56.3	18.4	VulRepair+GT	61.1	22.8
VRepair+Ours	42.0	11.9	VulRepair+Ours	44.0	15.0

4.5.3 *RQ3. Can the natural language suggestion help improve the effectiveness of patch generation for AVR tools?*

As discussed in Section 2.2, our VulAdvisor can not only guide developers to fix vulnerabilities, but can also used as additional information for automatic patch generation. To evaluate this, we utilize the ground-truth suggestions or predicted ones from VulAdvisor and concatenate them with the vulnerable function. We take VRepair [6] and VulRepair [12] as the baseline models to assess the impact of incorporating suggestions in vulnerability patch generation. Note that we do not explicitly mark the lines that require fixing during the fine-tuning process to simulate a real-world repair scenario. Instead, we take the original function as input and expect the model to generate the patch in the form of code changes (e.g., "*- unsigned int size; + size_t size;*"). To keep consistence with previous work, we use BLEU and Exact Match (EM) as the metrics.

Table 3 presents the results of this experiment. The table includes two settings: one with ground-truth suggestions (+GT) and the other with predicted suggestions from VulAdvisor (+Ours). For comparison, we also include the baseline results without any additional suggestions.

It is evident that the suggestions enhance patch generation performance across both underlying models. For instance, the base model VulRepair achieves moderate performance with BLEU and EM of 39.6% and 9.9%, respectively. When incorporating ground-truth suggestions into the fine-tuning process, we observe a significant improvement in all evaluation metrics. For example, the improvement in terms of BLEU is over an absolute value of 20%. This substantial boost can be attributed to the high-quality and accurate nature of the ground-truth suggestions, which provide precise information on how to fix the vulnerabilities. By incorporating these reliable suggestions during fine-tuning, VulRepair can better capture the necessary information and generate more effective patches. When comparing improvements over VRepair, the gains achieved with VulRepair are generally higher, indicating that our approach could be more effective for LLM-based AVR tools.

On the other hand, when using predicted suggestions from VulAdvisor (VulRepair+Ours), we still observe significant improvements over the baseline, although the performance is not as high as when using ground-truth suggestions. This is because that, while the predicted suggestions from VulAdvisor are helpful in guiding the patch generation process, they may not always be as precise as the ground-truth suggestions. The generated suggestions from VulRepair+Ours may contain minor errors or deviations from the optimal fixes, leading to slightly lower performance compared to VulRepair+GT. Nonetheless, our approach presents an effective and innovative strategy for enhancing AVR tools by integrating human-readable suggestions from LLMs.

Table 4: The score distribution of generated suggestions

Score	1	2	3	4	5	6	7	≥ 6	Avg
ChatGPT	45	20	16	13	5	1	0	1	2.16
Seq2seq	28	10	20	14	16	9	3	12	3.19
CodeBert	25	13	17	13	14	11	7	18	3.39
VulAdvisor	11	3	15	19	18	16	18	34	4.50

In conclusion, both ground-truth and predicted suggestions from VulAdvisor significantly enhance the effectiveness of patch generation for vulnerability repair. Ground-truth suggestions provide more precise and accurate guidance, resulting in the highest performance improvements. However, even with predicted suggestions, they still outperform the baselines, indicating that VulAdvisor's generated suggestions offer useful information. These findings demonstrate the high potential for improving automatic patch generation tools.

4.6 Human Evaluation

We conduct human evaluation for measuring the semantic correctness of generated suggestions and the utility of them for writing patches.

4.6.1 Semantic Relevance. We invited 6 PhD students, each with 2-5 years of software security experience, to participate in the evaluation. We compared our proposed approach VulAdvisor with three representative baselines, namely ChatGPT, Seq2seq, and CodeBert, from each of the categories in Table 1.

Specifically, we randomly selected 100 function-suggestion pairs from the testing set. We fetched the vulnerable function with the *diff* and oracle to facilitate assessing the generated suggestions, and evenly divided them into two groups. Each group of suggestions was then assigned to three different evaluators. The generated suggestions were presented to the evaluators in random order, so they were not aware of which approach the suggestion came from. Evaluators were asked to rate the similarity of generated suggestions to the references on a scale of 1 to 7. A score of 1 indicated "Not Similar At All", while a score of 7 indicated "Exactly The Same Meaning". Although a standard Likert scale typically ranges from 1 to 5 [34, 64], the 7-point scale is also widely used in existing studies [63, 65]. We chose the 7-point scale because it allows for a more fine-grained evaluation of the generated suggestions. Given the multiple levels of correctness involved, such as the accuracy of the vulnerable elements, the repair steps, and the explanations, a finer granularity helps us better differentiate between the strengths and weaknesses of the tools. For each suggestion, we collected three scores from evaluators and calculated the median value as the final score.

The results from Table 4 indicate that ChatGPT's performance is notably lower than its counterparts, as indicated by its highest concentration of suggestions rated at the lowest score (1) and the lowest average score of 2.16. This, again, could blame to the general nature of its security practice guidance and a potential gap in effectively mapping vulnerability semantics to appropriate solutions.

While the baselines outperform ChatGPT, their overall effectiveness is inadequate. Many of their scores are low, and a significant number of suggestions receiving scores of 1 or 2. As a result, the average score for the baselines is less than 3.5. In contrast, our proposed approach VulAdvisor outperforms these baselines significantly. For instance, it achieves 22 more high scores (score ≥ 6) compared to Seq2seq and 16 more compared to CodeBert. To statistically validate the improvements achieved by our approach, we conducted Wilcoxon signed-rank tests [54] on the 100 final scores. The p-values of the tests at a 95% confidence level are 1.2e-13, 2.8e-10 and 8.2e-07 respectively, when comparing our approach with Seq2seq and CodeBert. The results indicate that the improvements achieved by our approach are statistically significant. To sum up, the results of human evaluation confirm the effectiveness of the proposed approach.

4.6.2 Suggestion Utility. We conducted a controlled experiment to further assess the effectiveness of VulAdvisor. Similar to existing work [47, 61], we enlisted the help of 6 experienced developers from a security company. The participants were divided into two groups: Group 1 (G1) and Group 2 (G2), tasked with individual attempts to fix a vulnerable function with and without the assistance of VulAdvisor, respectively. We set the time limit to two days for the 100 samples, each of which could get 0-3 submissions for G1 and G2. For each sample within each group, we evaluated the first correct patch submitted, if any. After verification, we collected the recorded time spent on each successful repair.

We find that, without the suggestions, G2 produced correct patches for only 7 out of 37 samples that received submissions. In contrast, G1, with the aid of suggestions, submitted patches for 92 samples, 20 of which were successfully fixed. On average, G2 spent 25 minutes successfully fixing each vulnerability, whereas G1 only spent 8 minutes. These findings underscore the utility of our tool in facilitating the vulnerability repair process.

4.7 Case Study

We present two cases of vulnerable functions with their references and generated suggestions to discuss the superiority and limitation of our model. To facilitate understanding the vulnerable function, we additionally show the code change for fixing the vulnerability in it. Note that the input for each model is only the vulnerable function.

The first case shows a vulnerability with moderate complexity. ChatGPT's response, while somewhat related to the repair, mistakenly identifies the unrelated element for inspection. Similarly, the baseline models generate irrelevant suggestions. In contrast, our approach demonstrates remarkable accuracy in guiding the repair process. Notably, VulAdvisor precisely identifies the vulnerable element 'render frame', and correctly points to the fix position before the class 'AutofillHostMsg_ShowPasswordGenerationPopup'. This owes to the localness enhancement, which captures local features specific to this kind of function. Moreover, VulAdvisor provides clear instructions on how to rectify the issue. While it doesn't replicate the exact wording of the reference, the suggested meaning aligns closely. This effectiveness is largely due to the adopted repair action loss, which adjusts the sequence of generated words to ensure overall coherence.

While we have made some progress, the task of automatically generating high-quality repair suggestions remains a significant

Table 5: Two cases of vulnerable C/C++ functions with the repair suggestions. The pink highlights are the vulnerable elements, while yellow and blue highlights indicate the repair practice for the vulnerability, and the reason behind it respectively.

CASE 1	CASE 2

`void PasswordGenerationAgent::ShowGenerationPopup() {`
`+   if (!render_frame())`
`+     return;`
` Send(new AutofillHostMsg_ShowPasswordGenerationPopup(`
`       routing_id(),`
`       render_frame()->GetRenderView()->`
`            ElementBoundsInWindow(`
`           generation_element_),`
`       generation_element_.maxLength(),`
`       generation_element_.nameForAutofill(),`
`       is_manually_triggered_,`
` *generation_form_data_->form));`
`  generation_popup_shown_ = true;`
`}`
	`cdf_count_chain(const cdf_sat_t *sat, cdf_secid_t sid,`
	`       size_t size)`
	`{`
	`- cdf_secid_t maxsector=(cdf_secid_t)(sat->sat_len *`
	`      size);`
	`+ cdf_secid_t maxsector=(cdf_secid_t)((sat->sat_len *`
	`      size)/sizeof(maxsector));`
	`     ...`
	`- if (sid > maxsector) {`
	`-   DPRINTF(("Sector %d > %d\n", sid, maxsector));`
	`+ if (sid >= maxsector) {`
	`+   DPRINTF(("Sector %d >= %d\n", sid, maxsector));`
	`  }`
	`  sid = CDF_TOLE4((uint32_t)sat->sat_tab[sid]);`
	`} //The complete function is available at our website.`

Reference: Add an additional check for the render_frame() function before sending a message to show the password generation popup. This ensures that the function is not called if the render_frame() function returns false, preventing any potential errors or vulnerabilities.	**Reference**: Add a division operation to calculate the maximum sector size based on the size of the sector and the length of the SAT table. Additionally, the comparison operator can be changed to include equality, ensuring that the sector ID is not greater than or equal to the maximum sector size. This prevents the function from accessing memory outside the bounds of the SAT table.
ChatGPT: Sanitize the input and validate the data before sending it. This can be done by implementing input validation checks such as checking for null values, ensuring the maxLength is within acceptable limits, and validating the nameForAutofill field. Additionally...(*more but irrelevant words*)	**ChatGPT**: Add a check to ensure that the loop does not exceed a certain limit, such as CDF_LOOP_LIMIT. If the loop limit is reached, set errno to EFTYPE and return -1. Additionally, you can add a check to ensure that the sid value does not exceed the maximum sector...(*more but irrelevant words*)
Seq2seq: Add a check for the pf_exiting function call, which checks if the cloned does not exceed the available channel.	**Seq2seq**: Change the DCHECK_EQ function to limit the read compilations structure (contributions) and adding a new zVal which trap function is executed...(*more but irrelevant words*)
CodeBert: Replace the creation of a generator with a call to the AnimationStart() function, which returns a pointer to the animation object. This ensures that the object is properly displayed and prevents any potential memory corruption issues.	**CodeBert**: Change the data type of the variable i from int to long to prevent integer overflow. Additionally, a check can be added to ensure that the loop does not exceed the maximum allowed value.
VulAdvisor: Add a check for the existence of the render frame before creating a new AutofillHostMsg_ShowPasswordGenerationPopup message. If the render frame does not exist, the function returns without executing any further code.	**VulAdvisor**: Add a new variable maxsector which is calculated based on the value of sat->sat_tab[sid]. If sid is greater than maxsector, the function returns an error. If sid is equal to 0, the function returns 0. This prevents the function from accessing memory outside the bounds of the sat_tab array and potentially causing a buffer overflow.

challenge. For instance, in Case 2, nearly all methods, including ours, struggle to offer effective repair guidance for the buffer overflow vulnerability. ChatGPT does mention relevant elements like 'sid', but falls short in providing actionable repair steps. Despite VulAdvisor's apparent advantage in pinpointing vulnerable elements and providing a clear explanation, it does not succeed in generating a precise solution. This limitation might blame to the lack of external dependency information, such as the definition of 'maxsector'. Also, our approach's insensitivity to variable values sometimes leads to inaccurate predictions. These observations underscore the necessity of incorporating global dependency and dynamic information in future. We hope our work will inspire and draw more research attention to tackle this challenging problem.

5 RELATED WORK

5.1 Automated Vulnerability Repair

Due to the necessity of repairing vulnerabilities, many patch generation techniques have been proposed. Apart from various traditional ones [8, 13, 17, 25, 29, 68], there are arising DL-based Automated Vulnerability Repair (AVR) approaches in the literature [6, 7, 12]. For example, Chen et al. [6] introduced VRepair, an AVR approach based on transfer learning to deal with the problem of insufficient vulnerability data.

Very recently, LLM-based AVR has attracted much more attention. VulRepair [12] proposed the first approach that leverages an LLM, T5 architecture, on a large code corpus for patch generation.

Additionally, LLMs have been explored for zero-shot vulnerability repair [40]. However, when evaluated on real-world scenarios, these LLMs struggled to generate correct fixes, indicating that they are not yet ready to provide real-world value. Furthermore, a recent study compared the repair capabilities of LLMs and DL-based APR models in the context of Java vulnerabilities [57]. The work evaluated five LLMs and four DL-based APR techniques on real-world Java vulnerability benchmarks. It was found that existing LLMs and APR models had limited success in fixing Java vulnerabilities.

Different from the above studies that focus on patch generation for vulnerability repair, we open a new dimension to facilitate developers with readable suggestions in the form of natural language for ease of the process.

5.2 Automated Program Repair

At the same time, there has been a surge in the development of APR techniques built upon DL [21, 27, 32, 60, 70], , as well as those extended to repair deep neural networks [26, 45]. The advent of LLMs has opened new opportunities for the APR community. For instance, Yuan et al. [62] introduced CIRCLE, a program repair framework powered by T5, offering continuous learning capabilities across multiple programming languages. Xia et al. [59] presented AlphaRepair, a cloze-style APR approach based on CodeBERT, without the need for fine-tuning with historical bug-fixing data. Jiang et al. [20] investigated the effectiveness of pre-trained models, both with and without fine-tuning, in the program repair domain. Additionally, Xia et al. [58] conducted an extensive evaluation of recent pre-trained models for rectifying issues in real-world projects. Their findings demonstrated that state-of-the-art pre-trained models, including CodeX, were capable of effectively addressing a substantial number of bugs. Huang et al. [16] conducted a comprehensive study on the program repair capability of LLMs in the fine-tuning paradigm, which significantly outperforms previous state-of-the-art APR tools. Still, all these work focuses on generating bug patches instead of natural language suggestions for vulnerability repair.

6 THREATS TO VALIDITY

Internal Validity. The quality of our constructed dataset, including the functions and the ground-truths (i.e., suggestions), may affect the effectiveness of our approach. To mitigate it, we have performed the quality assessment using a sampling method that ensures 95% confidence, and the results demonstrate its high quality. Nevertheless, we cannot guarantee that all the samples are of equal quality. We will consider better evaluation metrics or new methods for constructing our dataset in the future. For our approach VulAdvisor, we did not fully explore the role played by the hyper-parameters on its performance. This is because fine-tuning LLMs can take a long time while there are many combinations of hyper-parameters. Furthermore, the identification of repair actions may not be optimal due to its heavy reliance on domain knowledge and semantic understanding. Nevertheless, our extensive evaluation has demonstrated its high potential. We encourage further research to refine and enhance this strategy in the future.

External Validity. We have evaluated the effectiveness of our approach in real-world projects by addressing RQ1-RQ3. However, our findings are currently limited to the C/C++ programming languages. In our future work, we plan to extend the diversity of vulnerabilities. Regarding the baselines, we have considered a variety of popular models. Models with even larger sizes (e.g., 13B), could be potential candidates for vulnerability repair suggestions. However, this requires substantial computational resources that are currently beyond our reach.

7 CONCLUSION

In this paper, we propose VulAdvisor, an LLM-based approach for automatically generating developer-centric suggestions towards vulnerability repair. Unlike existing methods, VulAdvisor provides natural language suggestions with detailed steps and reasons. We leverage the power of ChatGPT and CodeT5 for oracle extraction and suggestion learning. By incorporating the localness and repair action, VulAdvisor excels in generating high-quality vulnerability fix suggestions. Compared to baselines, VulAdvisor demonstrates superior performance and meanwhile enhances patch generation. The extensive experiments and human feedback demonstrate the effectiveness of VulAdvisor. Our source code and experimental data are publicly available at https://github.com/zhangj111/VulAdvisor.

ACKNOWLEDGEMENTS

This research/project is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG2-100E-2023-119), the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), and the Ministry of Education, Singapore under its Academic Research Fund Tier 1 (RG96/23). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, Cyber Security Agency of Singapore and Ministry of Education, Singapore.

REFERENCES

[1] 2023. Common Vulnerabilities and Exposures. https://cve.mitre.org/.
[2] 2023. The Heartbleed Bug. https://heartbleed.com/.
[3] 2023. National Vulnerability Database. https://nvd.nist.gov/.
[4] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4998–5007.
[5] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 117–128.
[6] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering* 49, 1 (2022), 147–165.
[7] Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. 2022. Seqtrans: automatic vulnerability fix via sequence to sequence learning. *IEEE Transactions on Software Engineering* 49, 2 (2022), 564–585.
[8] Weidong Cui, Marcus Peinado, Helen J Wang, and Michael E Locasto. 2007. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *2007 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 252–266.
[9] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 422–431.
[10] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[12] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 935–947.

[13] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–27.

[14] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher Reale, Rebecca Russell, Louis Kim, et al. 2018. Learning to repair software vulnerabilities with generative adversarial networks. *Advances in neural information processing systems* 31 (2018).

[15] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*. 200–210.

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

[17] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 539–554.

[18] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2022. The secret life of software vulnerabilities: A large-scale empirical study. *IEEE Transactions on Software Engineering* 49, 1 (2022), 44–63.

[19] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. *arXiv preprint arXiv:2302.05020* (2023).

[20] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1430–1442. https://doi.org/10.1109/ICSE48619.2023.00125

[21] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[22] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 135–146.

[23] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

[24] Ivan Victor Krsul. 1998. *Software vulnerability analysis*. Purdue University.

[25] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. Memfix: static analysis-based repair of memory deallocation errors for c. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 95–106.

[26] Tianlin Li, Yue Cao, Jian Zhang, Shiqian Zhao, Yihao Huang, Aishan Liu, Qing Guo, and Yang Liu. 2024. RUNNER: Responsible UNfair NEuron Repair for Enhancing Deep Neural Network Fairness. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[27] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 602–614.

[28] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[29] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. 2007. AutoPaG: towards automated software patch generation with source code root cause identification and repair. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. 329–340.

[30] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE conference on software testing, validation and verification (ICST)*. IEEE, 102–113.

[31] Zheheng Luo, Qianqian Xie, and Sophia Ananiadou. 2023. Chatgpt as a factual inconsistency evaluator for abstractive text summarization. *arXiv preprint arXiv:2303.15621* (2023).

[32] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

[33] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H Deng. 2017. Vurle: Automatic vulnerability detection and repair by learning from examples. In *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22*. Springer, 229–246.

[34] Parvez Mahbub, Ohiduzzaman Shuvo, and Mohammad Masudur Rahman. 2023. Explaining software bugs leveraging code structures in neural machine translation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 640–652.

[35] Joshua Maynez, Shashi Narayan, Bernd Bohnet, and Ryan McDonald. 2020. On Faithfulness and Factuality in Abstractive Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 1906–1919.

[36] Fairuz Nawer Meem, Justin Smith, and Brittany Johnson. 2024. Exploring Experiences with Automated Program Repair in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–11.

[37] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering*. 1169–1180.

[38] OpenAI. 2022. *Introducing ChatGPT*. https://openai.com/blog/chatgpt

[39] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[40] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.

[41] Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th international conference on Software engineering*. 390–401.

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

[43] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X Liu. 2012. A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 771–781.

[44] Chenglei Si, Zhe Gan, Zhengyuan Yang, Shuohang Wang, Jianfeng Wang, Jordan Lee Boyd-Graber, and Lijuan Wang. 2022. Prompting GPT-3 To Be Reliable. In *The Eleventh International Conference on Learning Representations*.

[45] Xinyu Sun, Wanwei Liu, Shangwen Wang, Tingyu Chen, Ye Tao, and Xiaoguang Mao. 2024. AutoRIC: Automated Neural Network Repairing Based on Constrained Optimization. *ACM Transactions on Software Engineering and Methodology* (2024).

[46] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.

[47] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically generated patches as debugging aids: a human study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 64–74.

[48] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant–How far is it? *arXiv preprint arXiv:2304.11938* (2023).

[49] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. 2022. What makes a good commit message?. In *Proceedings of the 44th International Conference on Software Engineering*. 2389–2401.

[50] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 269–280.

[51] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. 2020. Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In *29th USENIX Security Symposium (USENIX Security 20)*. 109–126.

[52] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. RAP-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 146–158.

[53] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[54] Frank Wilcoxon, SK Katti, and Roberta A Wilcox. 1970. Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test. *Selected tables in mathematical statistics* 1 (1970), 171–259.

[55] Emily Winter, David Bowes, Steve Counsell, Tracy Hall, Sæmundur Haraldsson, Vesna Nowack, and John Woodward. 2022. How do developers really feel about bug fixing? directions for automatic program repair. *IEEE Transactions on Software Engineering* (2022).

[56] Emily Rowan Winter, Vesna Nowack, David Bowes, Steve Counsell, Tracy Hall, Sæmundur Haraldsson, John Woodward, Serkan Kirbas, Etienne Windels, Olayori McBello, et al. 2022. Towards developer-centered automatic program repair:

findings from Bloomberg. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1578–1588.

[57] 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* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1282–1294. https://doi.org/10.1145/3597926.3598135

[58] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*.

[59] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 959–971.

[60] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering*. 1506–1518.

[61] Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 740–751.

[62] Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. 2022. CIRCLE: Continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 678–690.

[63] Jian Zhang, Shangqing Liu, Xu Wang, Tianlin Li, and Yang Liu. 2023. Learning to Locate and Describe Vulnerabilities. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 332–344.

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

[65] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Yanjun Pu, and Xudong Liu. 2020. Learning to handle exceptions. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 29–41.

[66] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter Liu. 2020. Pegasus: Pre-training with extracted gap-sentences for abstractive summarization. In *International Conference on Machine Learning*. PMLR, 11328–11339.

[67] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. BERTScore: Evaluating Text Generation with BERT. In *International Conference on Learning Representations*.

[68] Yuntong Zhang, Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. 2022. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 691–702.

[69] 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*. 1–13.

[70] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 341–353.