Out of Sight, Out of Mind: Better Automatic Vulnerability Repair by Broadening Input Ranges and Sources

Xin Zhou
Singapore Management University
Singapore
xinzhou.2020@phdcs.smu.edu.sg

Kisub Kim*
Singapore Management University
Singapore
kisubkim@smu.edu.sg

Bowen Xu
North Carolina State University
USA
bxu22@ncsu.edu

DongGyun Han
Royal Holloway, University of London
United Kingdom
donggyun.han@rhul.ac.uk

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

ABSTRACT

The advances of deep learning (DL) have paved the way for automatic software vulnerability repair approaches, which effectively learn the mapping from the vulnerable code to the fixed code. Nevertheless, existing DL-based vulnerability repair methods face notable limitations: 1) they struggle to handle lengthy vulnerable code, 2) they treat code as natural language texts, neglecting its inherent structure, and 3) they do not tap into the valuable expert knowledge present in the expert system. To address this, we propose VulMaster, a Transformer-based neural network model that excels at generating vulnerability repairs by comprehensively understanding the entire vulnerable code, irrespective of its length. This model also integrates diverse information, encompassing vulnerable code structures and expert knowledge from the CWE system. We evaluated VulMaster on a real-world C/C++ vulnerability repair dataset comprising 1,754 projects with 5,800 vulnerable functions. The experimental results demonstrated that VulMaster exhibits substantial improvements compared to the learning-based state-of-the-art vulnerability repair approach. Specifically, VulMaster improves the EM, BLEU, and CodeBLEU scores from 10.2% to 20.0%, 21.3% to 29.3%, and 32.5% to 40.9%, respectively.

1 INTRODUCTION

As the software landscape expands rapidly [22], the number of software vulnerabilities has increased correspondingly [26]. According to Common Vulnerabilities and Exposures [13], the yearly discovery of vulnerabilities in 2022 sets a new record with 26,448 software vulnerabilities reported, marking a notable 59% increase compared to 2021 [64]. Recently, many automatic vulnerability prediction approaches [8, 51, 85] are proposed to identify the vulnerability. However, when a software vulnerability is detected, it becomes of utmost importance to promptly rectify and resolve it in order to minimize the potential risks of exploitation. However, addressing software vulnerabilities often requires specialized expertise, and the existing pool of experienced developers is insufficient to tackle the extensive number of vulnerabilities found in millions of software systems [31]. Furthermore, the manual resolution of vulnerabilities is a time-consuming process; the GitHub 2020 security report finds that it takes 4.4 weeks to fix a vulnerability after its identification [17]. As a result, there is an urgent demand for automated methods to repair vulnerabilities.

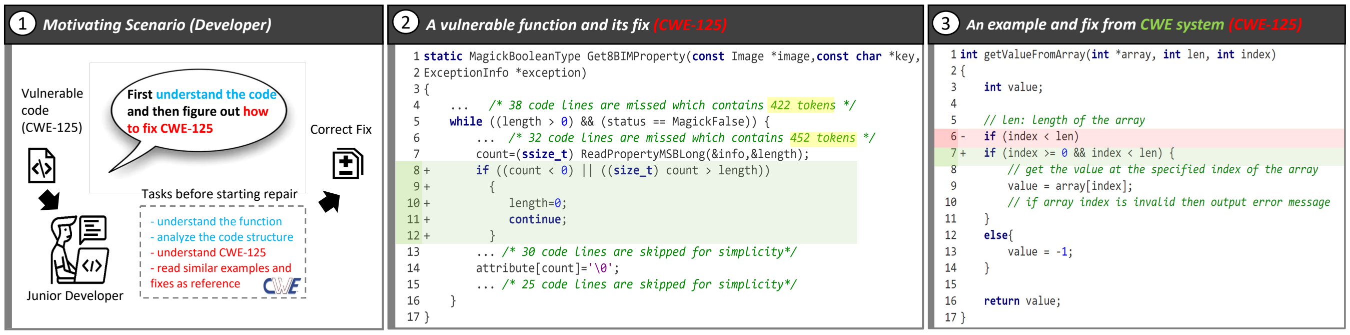
A number of program analysis-based vulnerability repair approaches [20, 21, 27, 29, 41, 46, 60, 66, 80] have been proposed. Although effective, program analysis-based approaches are often tailored for specific vulnerabilities or not applicable to certain types. In contrast, learning-based vulnerability repair approaches are not constrained to fixing specific vulnerability types. In this study, we aim to advance learning-based vulnerability repair approaches. Recently, several learning-based automatic vulnerability repair (AVR) approaches that take a vulnerable function with its CWE type (e.g., CWE-119) as input and generate the repaired function as output have been proposed [9, 19]. For instance, VRepair [9] is pre-trained on a large bug-fixing corpus and then trained to repair vulnerabilities. On the other hand, VulRepair directly utilizes a pre-trained model, named CodeT5 [71]. By harnessing this off-the-shelf pre-trained model, VulRepair achieves state-of-the-art performance in learning-based vulnerability repair [19]. Despite the promising outcomes, VRepair and VulRepair encounter three major **challenges**.

Understanding the entire vulnerable code: VRepair and VulRepair rely on the Transformer model, and its computational cost significantly increases with longer inputs. The attention matrix computations (i.e., the primary calculations) within the Transformer model scale quadratically with the length of the input sequence [65]. As a result, Transformer-based models inevitably restrict the length of input code snippets, leading to truncation of longer parts. For example, VulRepair imposes a maximum limit of 512 code tokens. However, real-world vulnerable codes often exceed this limit: 44.9% of the vulnerable code in VulRepair's evaluation dataset [19] surpasses 512 tokens. Failing to provide and understand the entire vulnerable code to the model reduces the likelihood of accurately addressing the vulnerability.

Understanding the structures of the vulnerable code: Both VRepair and VulRepair treat code in a similar manner to natural

*Corresponding author.

ICSE '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639222


1 Motivating Scenario (Developer)
Vulnerable code (CWE-125)
First understand the code and then figure out how to fix CWE-125
Correct Fix
Tasks before starting repair
- understand the function
- analyze the code structure
- understand CWE-125
- read similar examples and fixes as reference
Junior Developer
2 A vulnerable function and its fix (CWE-125)
3 An example and fix from CWE system (CWE-125)

```
 1   static MagickBooleanType Get8BIMProperty(const Image *image,const char *key,
 2   ExceptionInfo *exception)
 3   {
 4       ...   /* 38 code lines are missed which contains 422 tokens */
 5       while ((length > 0) && (status == MagickFalse)) {
 6           ...  /* 32 code lines are missed which contains 452 tokens */
 7           count=(ssize_t) ReadPropertyMSBLong(&info,&length);
 8 +         if ((count < 0) || ((size_t) count > length))
 9 +           {
10 +              length=0;
11 +              continue;
12 +           }
13           ... /* 30 code lines are skipped for simplicity*/
14           attribute[count]='\0';
15           ... /* 25 code lines are skipped for simplicity*/
16       }
17   }
```

```
 1   int getValueFromArray(int *array, int len, int index)
 2   {
 3       int value;
 4
 5       // len: length of the array
 6 -     if (index < len)
 7 +     if (index >= 0 && index < len) {
 8           // get the value at the specified index of the array
 9           value = array[index];
10           // if array index is invalid then output error message
11       }
12       else{
13           value = -1;
14       }
15
16       return value;
17   }
```

Figure 1: A motivating example. ❶ the process of how junior developers repair vulnerability; ❷ a vulnerable function and its fixes from the `ImageMagick` project; ❸ a vulnerable example and its fixes from the CWE website.

language texts, disregarding the incorporation of structural information. Natural language possesses a loosely structured nature, enabling words to be arranged in different orders while maintaining grammatical correctness [35, 50]. In contrast, programming languages exhibit a higher level of structure. Neglecting code structures could reduce the probability of effectively resolving the vulnerability in both VRepair and VulRepair. Prior research has demonstrated the benefits of including code structures, such as the Abstract Syntax Tree (AST), in tasks like code clone detection [49, 79], code search [54], and bug comprehension [48]. Following these prior studies, we also include the AST as a part of the model's input to improve its grasp of the structural characteristics of code.

Leveraging the expert knowledge: Software vulnerabilities do not exist in isolation. The Common Weakness Enumeration (CWE) system [11] offers a comprehensive catalog of common software weakness types. By utilizing the CWE system, one may gain access to accurate descriptions, vulnerable code examples, and information on closely related vulnerabilities pertaining to specific vulnerability types. In order to tap into the expert knowledge provided by the CWE system, both VRepair [9] and VulRepair [19] incorporate CWE types into their input data. The CWE system, however, offers a wealth of expert knowledge that extends beyond CWE types. This includes CWE names and vulnerable code examples shared on the web pages. Our approach aims to better utilize this abundant knowledge within the CWE systems.

To tackle these challenges, we propose **VulMaster**, a novel automatic vulnerability repair approach that aims to process the entire vulnerable code, regardless of its length. It then generates fixes by integrating diverse information, including vulnerable code structures and expert knowledge from the CWE system. VulMaster consists of three major components. First, it follows the idea of the *Fusion-in-Decoder (FiD)* framework [30] to overcome the limitations associated with the input length of Transformer-based models. Second, to capture the structural aspects of the vulnerable code, VulMaster utilizes the AST as part of its input. Third, it extensively utilizes expert knowledge of the CWE system, including the vulnerability type name, typical vulnerable code examples, and extra information on other closely related vulnerabilities.

We evaluate VulMaster on a real-world C/C++ vulnerability dataset used in previous studies [5, 9, 15, 19], which consists of 5,800 function-level unique vulnerability fixes collected from 1,754 large-scale open-source software projects. The experimental results illustrate that VulMaster enhances the EM, BLEU, and CodeBLEU scores from 10.2% to 20.0%, 21.3% to 29.3%, and 32.5% to 40.9%, respectively, compared to the state-of-the-art learning-based vulnerability repair solution. In summary, our contributions are as follows:

- To the best of our knowledge, we are the first to introduce the FiD framework for the field of software engineering. More importantly, we initiate how it should be leveraged to effectively mitigate the input length limitations of Transformer-based models and integrated with other valuable aiding information.
- We discover more diverse and valuable information such as vulnerable code structures and expert knowledge for CWEs, which can boost the effectiveness of repairing vulnerability.
- We further identify and address a hidden label leakage issue in the datasets used in the prior work, which can contribute to a more precise evaluation of future studies.

2 PRELIMINARIES AND MOTIVATION

2.1 Motivating Example

In ❶ of Figure 1, we show an example to analyze how junior security engineers repair a vulnerable function of CWE-125 (e.g., the vulnerable code before fixing shown in ❷ of Figure 1) and explain our two motivations below.

(1) Understanding the vulnerable function is the first thing. Instead of directly crafting an appropriate fix, the security developer may first undertake a meticulous examination of the vulnerable function to acquire a comprehensive understanding of the code and its weaknesses. While human developers can manage to grasp lengthy vulnerable functions, it poses a challenge for automatic tools based on Transformer models. For instance, as depicted in ❷ of Figure 1, the vulnerable function from [1] consists of over 800 code tokens before the vulnerable code lines, surpassing the input limit (512 tokens) of the learning-based SOTA VulRepair. Consequently, VulRepair failed to generate an accurate fix.

(2) CWE knowledge is helpful in inspiring the repair. To enhance the understanding of software vulnerabilities, the junior security developer may refer to the name/description and typical vulnerable examples of CWE-125 on the CWE website. From the example shown in ❸, one typical cause of CWE-125 is forgetting to ensure the array index is not a negative number. Upon identifying this pattern, the developer can revisit the target vulnerable function in ❷: *Line 7* of the target function obtains a value for the variable

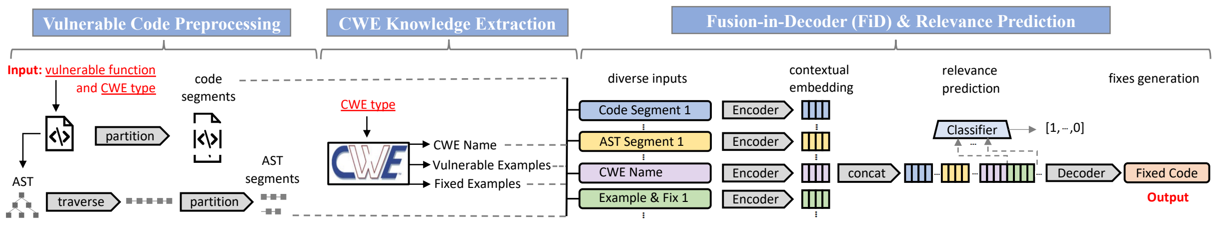


Figure 2: Overall Framework of VulMaster.

count, which is later used as an index in *Line 14*. However, the function does not check whether count is a negative value. As a result, a part of the ground-truth fix (in ❷ of Figure 1) could be inspired by the typical vulnerable example provided by the CWE website, i.e., checking whether the count is negative.

2.2 Background

Task Definition. In line with previous research [9, 19], learning-based automatic vulnerability repair (AVR) is formulated as a sequence-to-sequence problem: $(X_i, T_i) \longrightarrow Y_i$. Specifically, given a vulnerable code snippet X_i along with its associated CWE type T_i, a Deep Learning (DL) model generates the corresponding repaired code Y_i.

Fusion-in-Decoder (FiD) Framework is originally proposed for addressing challenges in the open-domain question-answering task in natural language processing (NLP) [30]. It is a sequence-to-sequence task, where the input is the question with numerous relevant passages and the output is the answer to the question. Before the introduction of FiD, researchers [42] concatenated the question and relevant passages to create the input for the encoder. However, the encoder had a limitation on the maximum input length, which led to the truncation of the input data (i.e., the concatenated questions and passages) and resulted in a substantial information loss.

The FiD framework adopts a divide-and-conquer approach to address such length limitation [30]. Specifically, each question and passage is individually encoded using an encoder, producing contextual embeddings. These contextual embeddings are subsequently concatenated to form a composite representation of the questions and all passages, which is then fed into a decoder to generate the desired output. By incorporating the FiD framework, Transformer-based models surpass the prescribed length limitation and gain the ability to effectively comprehend and encode numerous passages.

3 APPROACH

The framework of VulMaster is presented in Figure 2. VulMaster takes a vulnerable function and its CWE type as input and generates the corresponding fix. VulMaster involves three main parts, where the first two parts prepare the input data for the VulMaster model, and the third part is responsible for encoding the inputs and generating the fixed code.

Part 1: Vulnerable Code Preprocessing. Given a vulnerable function X_i, this part preprocesses the function into a code token sequence and the AST node sequence by traversing the ASTs.

Part 2: CWE knowledge Extraction. Given the CWE type T_i of the vulnerable function X_i, this part outputs the name, vulnerable code examples listed on CWE web pages, and the corresponding fixes for vulnerable code examples from CWE web pages.

Part 3: FiD and Relevance Prediction. This part first encodes all the input data (i.e., the code token sequences, AST node sequences, and CWE knowledge) into the contextual embeddings in a divide-and-conquer manner. Subsequently, the contextual embeddings are aggregated together. Then, the aggregated embedding is fed into the decoder to generate the corresponding repair Y_i.

Backbone model. VulMaster needs a pre-trained model as the backbone model and we employ CodeT5-base [71] for its impressive performance on various sequence-to-sequence tasks [53, 82].

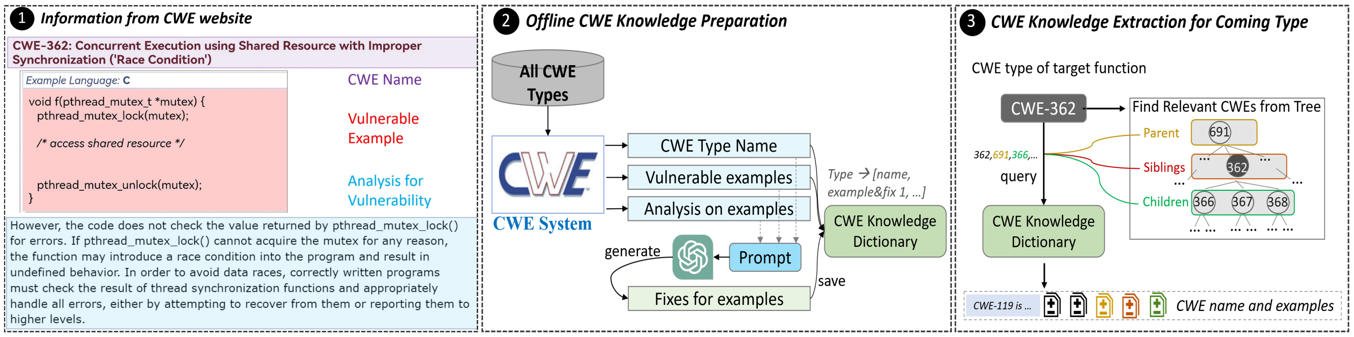
3.1 Vulnerable Code Preprocessing

This part aims to perform preprocessing on the target vulnerable functions to extract code segments and AST node sequences.

Tokenization. To correctly leverage the CodeT5 model, we utilize the CodeT5 tokenizer [72] to tokenize each vulnerable function into a token sequence.

Vulnerable function partitioning. VulMaster divides lengthy input code into multiple segments, ensuring that each segment adheres to the length limit (i.e., 512 tokens in the case of CodeT5). This helps VulMaster to further process and understand each of the code segments. Specifically, the process starts with tokenization, where the entire function is transformed into a sequence of tokens. VulMaster then proceeds to partition this token sequence into segments, ensuring that each segment contains no more than 512 tokens. If a function is shorter than 512 tokens, it constitutes a single segment.

AST node sequences. Existing approaches [9, 19] treat the input code as unstructured natural language text, disregarding the rich structure inherent in source code that can provide valuable semantic information. A notable structural representation of source code is the AST. Thus, we first adopt the tree-sitter parser to transform the input vulnerable function into the AST representation. However, pre-trained code models like CodeT5 are not designed to directly handle AST, as they are mainly optimized for sequential data rather than tree or graph structures. To address this limitation, we adopt a depth-first traversal method to convert the AST into an AST node sequence while preserving the structural information [28]. To generate the AST node sequence, we start by adding the concatenation of the *type* and *value* of the root node to the empty list A. The *value* refers to the actual token present in the source code, while the *type* represents the AST node's type. Next, we traverse the sub-trees of the root node in a depth-first order, adding the concatenation of values and types of the root nodes of each sub-tree into the list A. This recursive process is repeated for each sub-tree until all nodes within the tree have been traversed. Finally, the AST node sequence A is divided into multiple segments, each adhering to the length limit of 512 tokens.


1 Information from CWE website
CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Example Language: C
CWE Name
Vulnerable Example
Analysis for Vulnerability
2 Offline CWE Knowledge Preparation
All CWE Types
CWE System
CWE Type Name
Vulnerable examples
Analysis on examples
Type → [name, example&fix 1, ...]
CWE Knowledge Dictionary
generate
Prompt
save
Fixes for examples
3 CWE Knowledge Extraction for Coming Type
CWE type of target function
CWE-362
Find Relevant CWEs from Tree
Parent
691
Siblings
362
Children
366
367
368
362,691,366,...
query
CWE Knowledge Dictionary
CWE-119 is ...
CWE name and examples

```
void f(pthread_mutex_t *mutex) {
  pthread_mutex_lock(mutex);

  /* access shared resource */


  pthread_mutex_unlock(mutex);
}
```

However, the code does not check the value returned by pthread_mutex_lock() for errors. If pthread_mutex_lock() cannot acquire the mutex for any reason, the function may introduce a race condition into the program and result in undefined behavior. In order to avoid data races, correctly written programs must check the result of thread synchronization functions and appropriately handle all errors, either by attempting to recover from them or reporting them to higher levels.

Figure 3: Details of CWE knowledge extraction. ❶ the CWE name and one example from the CWE website; ❷ the process of generating fixes of typical vulnerable examples from CWE; ❷ the process to obtain vulnerable-fix code pairs given the target CWE.

3.2 CWE knowledge Extraction

This part involves gathering rich information from the CWE website for a CWE type of target vulnerable function.

Offline CWE knowledge preparation. To provide the CWE knowledge for an incoming vulnerable function instantly, we first build a CWE knowledge dictionary in an offline manner. The CWE knowledge dictionary is a dictionary where the key of each item is the CWE type and the corresponding value is a list of CWE knowledge, i.e., the CWE name, the vulnerable examples from CWE websites, and the fixes for those examples generated by ChatGPT. Figure 3 (❶) showcases the information available on the website of CWE-362, including the CWE name, the vulnerable examples from CWE websites, and the analysis of the examples written by experts. Note that the CWE website usually only provides vulnerable examples without fixed examples. Although vulnerable code examples exist, they do not possess the essential knowledge on "how to fix vulnerabilities". The analyses of the vulnerable examples contain the "how to fix" knowledge, but they are in natural language texts rather than source code. Therefore, we seek suitable fixes for the vulnerable code examples to complement the "how to fix" knowledge. To obtain the fixed code examples, we adopt the recent advancements in generative AI models, i.e., ChatGPT, which is proficient in generating code given detailed guidance [55, 70].

Figure 3 (❷) presents the overall process of building this offline CWE knowledge dictionary for VulMaster. Firstly, we begin by gathering all the CWE types that appeared in the experiment dataset [19]. Secondly, for each CWE type, we access its corresponding website to extract valuable information, i.e., the CWE name, typical vulnerable examples, and expert analysis of these examples. Thirdly, for each vulnerable code example and its accompanying analysis, we leverage ChatGPT to generate potential fixes. Specifically, we crafted the following prompt to generate the fixes:

"The code {*code*} contains a vulnerability of type {*name*}. The analysis of this vulnerable code is {*analysis*}. Please generate the repaired code to address the vulnerability:"

The {*code*}, {*name*}, and {*analysis*} are respectively filled in the vulnerable code example, CWE name, and expert analyses on this vulnerable example listed on the CWE web page. We utilize the prompt for each vulnerable example with its name and expert analysis and query ChatGPT (i.e., the `GPT-3.5-turbo` model with its default setting [55]) to obtain the fixed code. Given the ample and accurate expert analysis as guidance and the simplicity of those vulnerable examples from CWE, ChatGPT could generate a large number of correct fixes for those vulnerable examples from CWE web pages. We discuss such performance of ChatGPT for vulnerable code examples from CWE in Section 6.2.

However, different from the vulnerable code examples in CWE web pages, vulnerabilities in real-world software [19] often lack expert analyses, and the complexity of vulnerabilities in real-world software is significantly increased. Such factors make ChatGPT struggle to generate accurate fixes for real-world vulnerabilities, which is also discussed in Section 5.1.

CWE knowledge extraction for coming data. Given the CWE type of one target vulnerable function, VulMaster can easily access the CWE knowledge, including the CWE name, vulnerable code examples, and the generated fixes, by querying the CWE knowledge dictionary. In addition to extracting vulnerable examples and fixes for the target CWE type, VulMaster goes beyond this and includes examples and fixes from other closely related CWE types in the input. This process is represented as ❸ in Figure 3. The decision to include examples and fixes from closely related CWE types stems from one observation: CWE types are organized hierarchically, with parent, child, and sibling types sharing many similarities with a specific CWE type. For instance, CWE-125 represents "Out-of-bounds Read", while its children, such as "CWE-126: Buffer Over-read" and "CWE-127: Buffer Under-read", are more specific cases of CWE-125. Thus, VulMaster can consider a broader range of relevant code examples and fixes by incorporating examples and fixes from parent, child, and sibling CWE types (i.e., the wide range) in the input, rather than solely relying on those from the input CWE type (i.e., the narrow range).

3.3 Fusion-in-Decoder and Relevance Prediction

Backbone model adaption for AST. Although CodeT5 is pre-trained on a large number of source code snippets, it has not seen AST node sequences during its pre-training, resulting in a lack of familiarity with such structural representation. Prior studies [25, 81, 84] have suggested that the lack of pertinent software artifacts during pre-training may lead to suboptimal pre-trained models, as exemplified in pre-trained models for code changes [84], Android bytecode [62], and Stack Overflow posts [25]. A straightforward improvement involves incorporating the corresponding data during (additional) pre-training. Following this idea, we introduce an additional pre-training step to enhance the backbone model's understanding of AST structures, which requires a substantial and relevant code corpus. We choose to use the bug-fixing corpus provided by Chen et al. [9], consisting of over 500,000 pairs of buggy and fixed functions. The bug-fixing task shares similarities with vulnerability repair [9], making this corpus a valuable resource for us. Specifically, we further pre-train the CodeT5 model to generate the fixed version of buggy code. For half of the training samples, the inputs are AST node sequences of buggy code, while for the other half are the buggy code itself. Both the lengthy AST node sequences and source code are truncated to the first 512 tokens for simplicity. We also include the source code as a part of the inputs to ensure that the trained CodeT5 retains its ability to understand both the source code and AST semantics. In this model adaption phase, we employ the Adam optimizer [38] with a learning rate of 2e−5 and train CodeT5 for 5 epochs on the bug-fixing corpus. We denote the model after this step as *adapted-CodeT5*.

Contextual Embedding. As shown in Figure 2, VulMaster uses the encoder model to generate contextual embedding for the following input sources: 1) the complete vulnerable function X_i, 2) AST node sequences, 3) CWE type name, 4) examples and corresponding fixes of the CWE type T_i and related CWE types. Specifically, for one pair of vulnerable code example and its fix, we concatenate them into a single sequence and feed it to the encoder to get the contextual embedding. Notably, the encoder utilized is the encoder model of the *adapted-CodeT5*.

Relevance prediction. VulMaster is equipped with vulnerable-fixed code pairs gathered from web pages related to the input CWE type T_i (i.e., the CWE type of the input vulnerable function in the evaluation dataset) and its associated parent/child/sibling CWE types. However, not all vulnerable-fixed code pairs from CWE web pages hold the same level of relevance when it comes to repairing the input vulnerable function in the evaluation dataset [5, 15]. Pairs associated with the input CWE type are deemed the most relevant, while those linked to the input CWE type's parent/child/sibling CWE types are considered less relevant. The objective of the relevance prediction module is to assist VulMaster in identifying and highlighting the most relevant vulnerable-fixed code pairs from CWE web pages, while also considering less relevant pairs as contextual information. To achieve this, we introduce explicit supervision by performing a binary classification task [67].

Then we explain the preparation of the input for the relevance prediction module. The input is constructed by the following steps: Firstly, we collect vulnerable code snippets from the CWE web pages of the input CWE type T_i and its parent/child/sibling CWE types. Secondly, we collect the respective fixes (generated by ChatGPT) for the vulnerable code snippets from the first step. Thirdly, we concatenate each vulnerable code snippet from the first step with its corresponding fix, creating a single sequence. The resulting concatenated sequences (representing vulnerable-fixed code pairs from CWE web pages) are the input to the relevance prediction module. For the outputs (labels), we label each vulnerable-fixed code pair from CWE web pages as "most related" (class 1) if it belongs to the input CWE type T_i. Otherwise, it is labeled as "less related" (class 0).

The k-th vulnerable-fixed code pair from CWE webpages is transformed into its embedding E_k by the encoder model of VulMaster. Then the contextual embedding E_k is fed to a binary classifier (i.e., MLP [24]) to predict whether this k-th pair is "most related" or "less related". The output of the binary classifier is denoted as $p_k = \text{Classifier}(E_k)$. To update the model, we minimize the Cross-Entropy loss: $L_{relevance} = \sum_{k=1}^{K} -(g_k \log(p_k) + (1 - g_k) \log(1 - p_k))$. Here, g_k represents the ground truth relevance label of the k-th vulnerable-fixed code pair from CWE webpages and p_k is the prediction score of the pair. By optimizing this loss, a model can learn to effectively distinguish between "most related" and "less related" vulnerable-fixed code pairs from CWE webpages.

Fusion-in-Decoder. VulMaster combines all contextual embeddings of the input components into a single concatenated embedding, denoted as $C_{encoder}$. The concatenation is performed as follows: $C_{encoder} = [I_1, ..., I_n; A_1, ...A_m; D; E_1, ..., E_k]$ where ";" indicates the concatenation operation. I_j represents the j-th segment of the input vulnerable function, A_j represents the j-th segment of the AST node sequence of the input vulnerable function, D represents the CWE name and E_j represents the j-th example and its fix from CWE. The concatenated contextual embedding $C_{encoder}$ is then passed into the decoder model (i.e., the decoder of the *adapted-CodeT5*) to generate the fixed code. In general, the model is updated to minimize the loss: $L_{repair} = -log\ p(Y_i|X_i, AST_i, Name_i, Example\text{-}Fix\text{-}Pairs_i)$ This minimization aims to increase the probability of the model generating the correct fixed function (Y_i) using the provided diverse input components.

Multi-task Learning. Previous research has demonstrated that multi-task learning is beneficial in enhancing the performance of learning-based models in tasks such as code completion [44], code generation [69], and code understanding [68]. In the case of VulMaster, the adoption of the multi-task learning framework also aims to improve its effectiveness. Specifically, VulMaster is simultaneously trained on two tasks: 1) the relevance prediction task which identifies the most relevant vulnerable-fixed code pairs from CWE web pages from a pool of relevant pairs (i.e., $L_{relevance}$); 2) the vulnerability repair task which generates the fix for the input vulnerable function from the evaluation datasets [5, 15] (i.e., L_{repair}). The total loss function is the sum of the losses from each task: $L = L_{relevance} + L_{repair}$.

4 EXPERIMENTAL DESIGN

4.1 Dataset for Evaluation

We utilize the same real-world C/C++ vulnerability dataset that is used to evaluate the existing vulnerability repair approaches (i.e., VulRepair [19] and VRepair [9]). It consists of 8,482 pairs of vulnerable C/C++ functions and their corresponding fixes by merging two existing datasets: CVEFixes [5] and Big-Vul [15]. Please note that VulMaster is evaluated using the identical dataset combination of CVEFixes and Big-Vul, which was previously employed by the learning-based state-of-the-art VulRepair [19]. These pairs are collected from 1,754 open-source software projects spanning the period from 1999 to 2021. This dataset is divided into training (70%), testing (20%), and validation (10%) subsets by previous studies [19]. The data statistics are shown in Table 1. Notably, we checked and

Table 1: Statistics of the studied dataset

Dataset	Train	Valid	Test	All	%samples >512 tokens	%samples >1000 tokens
Original	5,937	839	1,706	8,482	45.6%	25.0%
Deduplication	3,872	316	1,612	5,800	44.9%	25.1%

confirmed that the vulnerable-fixed code pairs sourced from the CWE website (Section 3.2) have no duplicates with the vulnerability repair evaluation dataset [19] mentioned above.

Processing. To process the dataset, VulRepair [19] and VRepair [9] adopt the same steps, which involve the addition of special tokens. Particularly, each vulnerable function in the dataset is marked using the special tokens *<StartLoc>* and *<EndLoc>*. The *<StartLoc>* token indicates the beginning of the vulnerable code lines, while the *<EndLoc>* token indicates the end. For the ground truth, the special tokens *<ModStart>* and *<ModEnd>* are inserted into each repaired function to signify the start and the end of the repaired code lines, respectively. These special tokens serve the purpose of guiding the model's attention toward the vulnerable code lines and the corresponding fixed code lines. In other words, *<StartLoc>* and *<EndLoc>* provide the *perfect vulnerability localization* to vulnerability repair approaches.

Label leakage issue. In this paper, the reported performance of the learning-based SOTA VulRepir is different from its original paper [19]. This is caused by our leakage correction. The dataset used in VulMaster is merged from two existing datasets CVEFixes [5] and Big-Vul [15]. The authors of VulRepair seem to assume that data samples from two datasets are different from each other. By checking the vulnerable and fixed code pairs from each dataset, we found that about 60% of the pairs are duplicates. As the authors of VulRepair [19] further randomly split the merged dataset into training/validation/test sets, it leads to the presence of duplicated samples across the training, validation, and test sets. The label leakage issue might lead to an overestimation in performance measurement and artificially inflated scores [4]. To tackle this problem, we conducted a deduplication operation in three steps. Firstly, we removed any samples from the training set that were identical to any samples in the validation or test sets. Next, we eliminated any samples from the validation set that were identical to any samples in the test set. Finally, we eliminated 94 duplicate samples within the test set. The deduplication statistics are presented in Table 1.

To assess the impact of the label leakage issue, we first use the replication package [18] released by the authors of VulRepair to train VulRepair from scratch by using the original fine-tuning dataset. Experimental results indicate just a 0.2% difference between our replication and the reported performance in the VulRepair paper, validating the correct usage of their replication package. Next, we trained another VulRepair from scratch on the deduplicated version of the dataset, resulting in a decrease in VulRepair's performance from 44.2% to 10.2% in terms of Exact Match accuracy. This significant drop is expected because of the label leakage issue.

4.2 Baselines

We evaluate VulMaster against three groups of baseline models. The first group consists of learning-based automatic vulnerability repair approaches, namely VulRepair [19] and VRepair [9]. Both of these approaches take a vulnerable function concatenated with a CWE ID (e.g., CWE-119) as input and generate the fixed code as output. The second group comprises general-purpose pre-trained code models that are widely used in various code-related downstream tasks. Specifically, we include two encoder-based models (i.e., CodeBERT [16] and GraphCodeBERT [23]), two decoder-based models (i.e., PolyCoder-160M [77] and CodeGen-350M [52]), and two encode-decoder based models (i.e., CodeReviewer [43] and CodeT5-base [71]). Those models take the same input and output as VulRepair and VRepair. The third group consists of large language models (LLMs) for code, which have gained significant attention recently [3, 36, 76]. For LLMs, we include the well-known ChatGPT model (i.e., gpt-3.5-turbo) [55] and the improved version of ChatGPT model (i.e., gpt-4) [56] as baselines. We use a prompt similar to the prompt described in Section 3.2. We remove the sentence about expert analysis because of the lack of expert analyses in real-world vulnerabilities. Additionally, one sentence is added to explicitly tell LLMs about the meaning of special tokens (i.e., *<StartLoc>* and *<EndLoc>*). We also provide the CWE description and one vulnerable example from the CWE website in the prompt. The prompt used is: *"A vulnerability of type* $\{cwe\_type\}$ *refers to* $\{cwe\_description\}$*. One vulnerable example of this type is* $\{cwe\_example\}$*. The code* $\{code\}$ *contains a vulnerability of type* $\{cwe\_type\}$*. Note that <StartLoc> and <EndLoc> indicate the start and the end of vulnerable code lines. Please generate the repaired code to address the vulnerability:".*

4.3 Experimental Setting

Implementation details. Following prior work [9, 19], we also focus on fixing C/C++ vulnerabilities. Thus, we mainly collected C/C++ vulnerable examples from the CWE websites. But if there is no C/C++ vulnerable example for a CWE type, we will also collect one example of other languages like C# and Java. Please note that our approach is generic and language-agnostic and can be applied to any programming language. During training, we employ the Adam optimizer with a learning rate of $1e-4$. The weight decay rate is set to 0.01. The batch size is set to 64, and the training process consists of 20 epochs. After each epoch, we evaluate the model's performance on the validation set. The best checkpoint on the validation set is selected for the final testing. For the hyper-parameters of the FiD framework, the maximum length of an individual segment (e.g., a segment of a lengthy vulnerable function) is set to 512, aligning with the maximum length of CodeT5 [71]. Additionally, we observe that about 90% of the samples in our dataset comprise fewer than 10 segments, each containing up to 512 tokens. These segments encompass various input components, including the vulnerable function, AST node sequences, CWE descriptions, and examples from relevant CWE types. Therefore, we set the maximum number of segments of FiD to 10. We will discuss the impact of the choice of the maximum number of segments in Section 5.3 (RQ3).

Evaluation metric. In accordance with previous studies [19], we employ the Exact Match (EM) metric as our evaluation measure. EM refers to the percentage of the generated code that has the same token sequence as the ground truth. We also utilize another widely used metric, i.e. BLEU-4 [57] score, which evaluates the token-level similarity between the generated code and the ground truth. In addition, we utilize the CodeBLEU [58] score, which is a specialized variant of the BLEU score tailored for source code by additionally taking code structure into account. We consider the *Top 1 prediction* when calculating metrics.

Table 2: Model performance of baselines and VulMaster

Type	Approach	EM	BLEU	CodeBLEU
pre-trained	CodeBERT [16]	3.9	3.9	12.2
	+ bug-fixing and CWE data	7.3	6.3	22.0
	GraphCodeBERT [23]	3.6	2.0	9.9
	+ bug-fixing and CWE data	8.1	5.2	16.7
	PolyCoder [77]	3.5	4.3	9.9
	+ bug-fixing and CWE data	9.9	14.9	30.4
	CodeGen [52]	7.0	4.7	12.1
	+ bug-fixing and CWE data	12.2	17.4	30.3
	Codereviewer [43]	7.3	12.0	33.5
	+ bug-fixing and CWE data	10.2	13.0	38.0
	CodeT5 [71]	10.2	21.3	32.5
	+ bug-fixing and CWE data	16.8	24.2	35.3
LLM	GPT-3.5 [55]	3.6	8.8	17.6
	GPT-4 [56]	5.3	9.7	16.6
task-specific	VRepair [9]	8.9	11.3	31.8
	+ bug-fixing and CWE data	8.9	11.4	31.7
	VulRepair [19] (SOTA)	10.2	21.3	32.5
	+ bug-fixing and CWE data	16.8	24.2	35.3
Ours	VulMaster	**20.0**	**29.3**	**40.9**

5 EXPERIMENTAL RESULTS

Our work aims to answer three research questions (RQ).

- **RQ1: How effective is VulMaster compared to baselines?** In RQ1, we compare our VulMaster to learning-based SOTA vulnerability repair approaches, widely used pre-trained code models, and the latest LLMs.
- **RQ2: How do the key designs of VulMaster influence the model performance?** In RQ2, we conduct an ablation study to confirm the contributions of different modules.
- **RQ3: What are the influences of different design choices?** In RQ3, we investigate how different design choices impact the effectiveness of our VulMaster.

5.1 RQ1. The Effectiveness of VulMaster

Setup. We evaluate both the baselines (described in Section 4.2) and our VulMaster model using the dataset under investigation (discussed in Section 4.1). The evaluation metrics, namely EM, BLEU, and CodeBLEU, are detailed in Section 4.3. Notably, higher scores for all these metrics indicate better performance.

Moreover, in the development of VulMaster, we employed a bug-fixing dataset from [9] and vulnerable-fixed code pairs obtained from the CWE systems. However, the baseline models had not previously been fine-tuned on the same bug-fixing corpus or CWE data, potentially resulting in an unfair comparison. To mitigate this potential bias, we conducted additional experiments with the baselines, aligning them with our approach. This involved a four-step process: Firstly, we merged the bug-fixing dataset [9] with the vulnerable-fixed code pairs from the CWE system, creating a merged single dataset. Secondly, we conducted fine-tuning on this merged dataset for the baselines (with the exception of close-sourced GPT-3.5 and GPT-4). The objective of this fine-tuning process was to generate clean code snippets given the presence of buggy or vulnerable code. Thirdly, the baselines were further fine-tuned using the vulnerability repair dataset [5, 15] to prepare them for the vulnerability repair task. Finally, the baselines generated predictions for the test

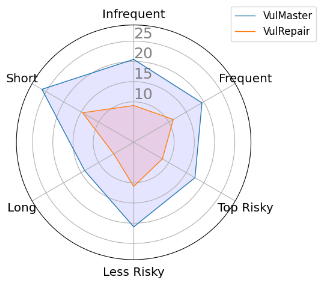


Figure 4: Model performance in EM on different groups.

set of the vulnerability repair dataset. The results achieved after enhancing the bug-fixing and CWE data in this manner are denoted by the label *"+bug-fixing and CWE data"*.

Results. Table 2 presents the performance comparisons between VulMaster and the baseline models, with the best results highlighted in bold. ***Our VulMaster achieves the best results among all baselines and outperforms the learning-based SOTA by a large margin.*** The experimental results illustrate that VulMaster enhances the EM, BLEU, and CodeBLEU scores from 10.2% to 20.0%, 21.3% to 29.3%, and 32.5% to 40.9%, respectively, compared to the state-of-the-art learning-based vulnerability repair approach VulRepair. We conducted the Wilcoxon signed-rank test [74] on the paired data between VulMaster and each of the baseline models. The resulting p-values for all the comparisons were found to be less than 0.001, indicating that the performance differences between VulMaster and the baseline models are statistically significant.

Furthermore, LLMs such as GPT-3.5 and GPT-4 struggle to effectively repair real-world software vulnerabilities possibly because they cannot update their parameters to fully utilize the training data. Sun et al. [63] also reported a similar phenomenon in code summarization, where GPT-3.5 performs notably worse than fine-tuned pre-trained CodeT5 in terms of BLEU scores.

Table 2 also highlights the improvements achieved by incorporating the bug-fixing corpus and CWE data into all baselines. Despite these improvements, VulMaster maintains a substantial lead over all baselines. Notably, VulMaster outperforms the best-performing baseline (the enhanced VulRepair) by a significant margin, with improvements of 19.0%, 21.1%, and 15.9% in terms of EM, BLEU, and CodeBLEU, respectively. One of the novel aspects of VulMaster is the usage of CWE data that was not leveraged in prior works. These experiment results demonstrate that the reason behind VulMaster's superior performance over the baselines is not the inclusion of the CWE data alone. Rather, the other unique designs of VulMaster, such as 1) the Fusion-in-Decoder module that handles lengthy vulnerable functions, and 2) the AST node sequence that provides vulnerable code structures to the model, are also beneficial.

Analyses. We further analyze the learning-based SOTA VulRepair (also the best-performing baseline) and our VulMaster. Specifically, we divided the whole test set into subgroups and observed how VulRepair and VulMaster perform in different subgroups. Those subgroups are split based on three different criteria: the lengths of the input functions, the frequencies of vulnerabilities, and the severity levels of vulnerabilities. In terms of the lengths of the input, two subgroups were created: *1) the long function group* consisted of half of the test samples whose vulnerable function lengths are

Table 3: Numbers of perfect predictions for the top-10 most dangerous CWEs

Rank	CWE-Type	Name	VulRepair	VulMaster	#sample
1	CWE-787	Out-of-bounds Write	3	12	58
2	CWE-79	Cross-site Scripting	0	1	1
3	CWE-89	SQL Injection	0	1	4
4	CWE-416	Use After Free	1	6	60
5	CWE-78	OS Command Injection	0	0	4
6	CWE-20	Improper Input Validation	18	33	128
7	CWE-125	Out-of-bounds Read	12	20	156
8	CWE-22	Path Traversal	0	0	5
9	CWE-352	Cross-Site Request Forgery	0	0	1
10	CWE-434	Dangerous File Type	-	-	0
		TOTAL	34 (8.2%)	73 (17.5%)	417

longer, and *2) the short function group*, consisted of the remaining test samples. The threshold between long/short groups is 449 tokens. In terms of the frequencies of vulnerabilities, we have: *3) the frequent group*, consisted of 50% of the test samples whose CWE types are more frequent, and *4) the infrequent group*, consisted of the remaining half of test samples. For the severity levels of vulnerabilities, we have: *5) the top risky group* consisted of test samples whose CWE types are one of the top 10 most dangerous CWE types, and *6) the less risky group* consisted of the remaining test samples.

Figure 4 shows the EM performance for each subgroup. ***VulMaster exhibits significant superiority over the learning-based SOTA in all investigated dimensions.*** For example, VulMaster could improve the EM scores from 8.2% to 17.5% for the top risk group. Table 3 presents the number of perfect predictions made by VulMaster for the top risk group. Although the correct fix ratio (17.5%) of VulMaster is not high enough, this improvement over the SOTA is significant and suggests that our method is a substantial step towards more practical automatic vulnerability approaches in the future. Additionally, we observe that VulRepair exhibits lower performance with infrequent vulnerabilities compared to frequent ones, consistent with findings from a recent study [83] that suggested learning-based approaches may struggle with infrequent data. In contrast, VulMaster achieves comparable performance across both frequent and infrequent vulnerabilities, demonstrating its great performance with less common data.

Answer to RQ1: VulMaster achieves the best performance (20.0% in EM) among all baselines. Specifically, VulMaster enhances the EM, BLEU, and CodeBLEU scores from 10.2% to 20.0%, 21.3% to 29.3%, and 32.5% to 40.9%, compared to the state-of-the-art learning-based approach.

5.2 RQ2. The Impact of the Core of VulMaster

Setup. This RQ aims to investigate the contributions of the key designs of our approach. Specifically, we conduct two sets of ablation studies using different backbone models: the input component designs and the DL model designs. In each ablation study, we remove one component at a time to examine the individual contributions of key designs. *(1) VulMaster w/o entire function*: removing the extended part of lengthy functions and only including the first 512 tokens, similar to the prior approaches like VulRepair, *(2) VulMaster w/o AST*: excluding the AST traversals, and *(3) VulMaster w/o CWE knowledge*: eliminating the CWE name, vulnerable examples from CWE, and generated fixes for CWE vulnerable examples by ChatGPT. For DL model designs, we experiment with three variants: *(4) VulMaster w/o relevance-classifier*: disregarding the binary classification task used to identify relevant vulnerable examples, *(5) VulMaster w/o FiD*: removing the FiD, resulting in the simple utilization of the backbone model (truncating inputs at the 512 tokens). *(6) VulMaster w/o model adaptation*: leveraging the original CodeT5-base rather than the *adapted-CodeT5* in Section 3.3.

Table 4: Ablation study on VulMaster

Type	Variants	EM
Full Model	VulMaster	**20.0**
Input Components	-w/o entire function	18.8
	-w/o AST	19.0
	-w/o CWE knowledge	18.9
Model Designs	-w/o relevance prediction	19.6
	-w/o FiD	16.7
	-w/o model adaptation	<u>13.6</u>

Results and Analyses. The experimental results are presented in Table 4, with the best results highlighted in bold and the largest performance drop in the underline. Our main findings are: ***1) All key designs are essential to achieve the best performance.*** Based on the results shown in Table 4, we observe that the absence of each key design in VulMaster leads to a reduction in the Exact Match score. It demonstrates that each component plays an important role in VulMaster. Specifically, the complete function, structural information, and CWE knowledge bring 6%, 5%, and 5.5% improvements, respectively. For DL model designs, the auxiliary relevance prediction task, the FiD, and the model adaptation bring 2%, 16.5%, and 32% improvements, respectively. ***2) The model adaptation is the most effective module.*** We find that incorporating model adaptation with the bug-fixing corpus contributes to a significant improvement in VulMaster. While Chen et al. [9] demonstrated the effectiveness of bug-fixing corpus in the vanilla Transformer model, we are the first to show its high efficacy in adapting pre-trained models like CodeT5.

Answer to RQ2: All designs are essential for the performance of VulMaster. Besides, our input and model designs are effective and improve the performance by at most 32% in EM.

5.3 RQ3. Influences of Design Choices

Setup. In this RQ, we experiment VulMaster with three main design choices: 1) the maximum number of segments, 2) the prompt used to generate fixes for vulnerable examples from the CWE system, and 3) the ChatGPT models used to generate the fixes for vulnerable examples. For the maximum number of segments (denoted as K), it is a key hyperparameter in FiD [30], which controls the improved range of inputs. For instance, if $K = 10$, then the maximum input range of VulMaster is broadened from 512 to 5,120 tokens. In this RQ, we conduct experiments by varying K from 1 to 20, with an increment of 5. This aims to validate our choice of $K = 10$. For prompts, they can influence the performance of LLMs, like ChatGPT, to some extent [6, 59]. While the prompt used in Section 3.2 may not be the optimal choice, it is impractical to explore all potential prompts [59]. To examine VulMaster's sensitivity to different

Table 5: Effectiveness and Efficiency when the maximum number of segments vary

Max. Number of Segments	1	5	10	15	20
Validation EM (%)	12.9	25.3	27.8	27.8	27.6
Testing EM (%)	10.2	19.8	20.0	20.1	19.9
Inference Time per Function (s)	0.7	1.2	1.9	2.8	4.1

prompts, we evaluate three distinct prompts, as shown in Table 6. P0 is the prompt we used in Section 3.2, P1 is the rephrased version of P0, and P2 is a simplified version of P0. This experiment aims to demonstrate that VulMaster can deliver satisfactory performance even with varied prompts. For ChatGPT models, OpenAI continuously and episodically updates its ChatGPT models [55]. To assess the performance of VulMaster to different versions of ChatGPT models, we experiment with three ChatGPT model versions: GPT-3.5-turbo, GPT-3.5-turbo-0301, and GPT-3.5-turbo-0613. Among them, GPT-3.5-turbo is the latest model. On the other hand, GPT-3.5-turbo-0301 and GPT-3.5-turbo-0613 are deprecated snapshots from March 1st and June 13th of 2023, respectively.

Results of Varying Maximum Number of Segments. Table 5 presents the experimental results. In previous experiments, we chose $K = 10$ as it adequately covered over 90% of the data samples. From Table 5, we further validate that ***K=10 strikes a balance between effectiveness and efficiency.*** On one hand, when K is less than 10, VulMaster's effectiveness drops accordingly. For example, with $K = 5$, the validation and testing EM scores decrease by 9.9% and 1.0%, respectively. On the other hand, when K is larger than 10, there is no further improvement in VulMaster's test EM scores. This is likely because $K = 10$ already covers all input segments of over 90% of data samples, and increasing it further does not provide significant benefits. However, higher K leads to slower inference. $K = 10$ reaches a balance between effectiveness and efficiency.

Results of Varying Prompts and ChatGPTs. The experimental results are shown in the bottom part of Table 6. For three different prompts and different ChatGPT, we observe that VulMaster performance varies from 19.8% to 20.3%, with at most 3% differences. This indicates that ***VulMaster can deliver satisfactory performance with varied prompts and ChatGPT models.***

Answer to RQ3: By setting the maximum number of segments as 10, VulMaster achieves a balance between effectiveness and efficiency. Besides, VulMaster shows stable performance across multiple plausible prompts and ChatGPT model versions, with at most 3% differences.

6 DISCUSSION

6.1 Case Study

Figure 5 presents two fixes generated by the SOTA VulRepair and VulMaster for a vulnerable function [37] of CWE-401 from the evaluation dataset. CWE-401 refers to situations where memory is allocated dynamically but not properly released after it is no longer needed, leading to memory leaks [12]. The vulnerable function waits for the completion of a prior operation (*Line 10*), and if the operation times out (*Line 12*), it returns with an error code (*Line 15*). However, it fails to release the dynamically allocated buffer (skb) before the return statement, resulting in a CWE-401 vulnerability.

Table 6: Different prompts and the performance of VulMaster when prompts and ChatGPT models vary

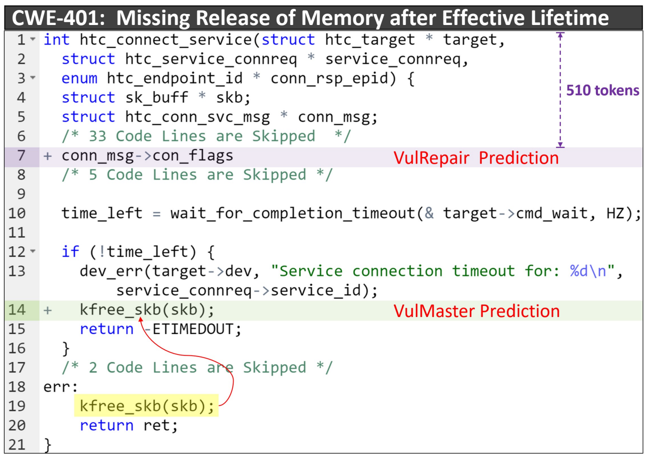
		Multiple Prompts		
Prompt Details	P0	The code {code} contains a vulnerability of type {name}. The analysis of this vulnerable code is {analysis}. Please generate the repaired code to address the vulnerability:		
	P1	The code {code} suffers from {name}. Here is the analysis for the code: {analysis}. Please output the repaired code to fix the vulnerability:		
	P2	{code} contains {name}. {analysis}. Please generate the repaired code to address the vulnerability:		
	EM	gpt-3.5-turbo-0301	gpt-3.5-turbo-0613	gpt-3.5-turbo (latest)
Results	P0	19.9	20.1	20.0
	P1	20.0	20.1	20.1
	P2	19.8	20.3	20.1

VulMaster correctly generates the fix (*Line 14*) to free the memory with skb. Its understanding of the entire function enables it to recognize the error-handling mechanism already present in *Line 19* (i.e., `kfree_skb(skb);`), helping it to choose the suitable function (i.e., `kfree_skb`) for freeing the memory. In contrast, the SOTA VulRepair makes a wrong prediction because the vulnerable function exceeds VulRepair's input limit of 512 code tokens before the vulnerability location (*Line 14*). As a result, VulRepair's prediction only adds an irrelevant code line (*Line 7*).

6.2 How Do Fixes of CWE Examples Help?

The ablation study (RQ2) revealed the effectiveness of the CWE knowledge. The potential reason behind the effectiveness lies in that the generated fixes do not only describe "what is vulnerable" like the CWE names and vulnerable examples, but also explicitly provide knowledge on "how to fix an identified vulnerability". This is crucial information for the vulnerability repair task. The validity of this explanation depends on the accuracy of the generated fixes.

To validate the explanation, the first and second authors manually assessed the correctness of the generated fixes for C/C++ vulnerable code examples. They carefully examined the vulnerable examples and the expert analyses from the CWE website and then reviewed the potential fixes generated by ChatGPT. Fixes that successfully mitigated the vulnerability were labeled as "correct," while those that did not were labeled as "wrong." In case of any



```
1   int htc_connect_service(struct htc_target * target,
2     struct htc_service_connreq * service_connreq,
3     enum htc_endpoint_id * conn_rsp_epid) {
4     struct sk_buff * skb;
5     struct htc_conn_svc_msg * conn_msg;
6     /* 33 Code Lines are Skipped  */
7 + conn_msg->con_flags
8     /* 5 Code Lines are Skipped */
9
10    time_left = wait_for_completion_timeout(& target->cmd_wait, HZ);
11
12    if (!time_left) {
13      dev_err(target->dev, "Service connection timeout for: %d\n",
          service_connreq->service_id);
14 +    kfree_skb(skb);
15      return -ETIMEDOUT;
16    }
17    /* 2 Code Lines are Skipped */
18  err:
19      kfree_skb(skb);
20      return ret;
21  }
```

Figure 5: Example of repairs by VulMaster and VulRepair.

Table 7: The numbers of correctly fixed vulnerabilities by LLMs, APR models, and our VulMaster in the Vul4J [7] benchmark

	Frozen LLMs					Fine-tuned LLMs				APR models				Ours
	Codex	CodeT5	CodeGen	PLBART	InCoder	CodeT5	CodeGen	PLBART	InCoder	CURE	Recoder	RewardR	KNOD	VulMaster
Vul4J (35)	6.2	2	1	0	3	2	5	2	6	1	0	0	1	**9**

discrepancies in annotations, a separate meeting was conducted to resolve each discrepancy, resulting in the final labels (shared in our replication package [2]). The manual investigation revealed that ChatGPT achieved an accuracy of **76.1% (67/88)** in generating fixes for C/C++ vulnerable examples from CWE. This high accuracy supports our explanation to some extent.

Additionally, ChatGPT performs exceptionally well in generating fixes for vulnerable examples from CWE. However, it struggles to repair vulnerabilities from real-world projects (RQ1). The reason is that security experts have already provided ample analyses and descriptions of the vulnerable examples on the CWE website (as depicted in ❶ of Figure 3). In contrast, such detailed information is often unavailable when repairing vulnerabilities in real-world projects, making the task much more challenging.

6.3 Generalizability to Benchmark with Tests

Motivation. In our previous experiments, we utilized evaluation data from CVEFixes [5] and Big-Vul [15], encompassing a wide range of vulnerabilities collected from 1,754 C/C++ projects. However, this evaluation data solely included vulnerable code snippets and their associated fixes, lacking accompanying test cases. To ensure VulMaster's generalizability to benchmarks that include test cases, we conducted an additional evaluation on the Vul4J [7] benchmark. Vul4J [7] is a high-quality Java vulnerability repair benchmark and includes test cases. Our evaluation of Vul4J also demonstrates VulMaster's versatility in handling multiple programming languages, such as C/C++ [5, 15] and Java [7].

Setup. The Vul4J benchmark serves as a test dataset for evaluation and does not provide a training pipeline for learning-based approaches. To overcome this limitation, we adopted the methodology of a recent study by Wu et al. [75], which conducted experiments comparing multiple Large Language Models (LLMs) and Automatic Patch Repair (APR) models on the Vul4J benchmark. Wu et al. [75] conducted their evaluation by specifically selecting 35 single-hunk vulnerabilities from the Vul4J benchmark, as the APR models under investigation were primarily designed to address single-hunk bugs. Their comparative analysis involved three main categories: 1) the frozen LLMs, 2) fine-tuned LLMs, and 3) state-of-the-art APRs. The frozen LLMs directly performed inferences on the test set. For the fine-tuned LLMs, Wu et al. addressed the limited availability of Java vulnerabilities for fine-tuning by utilizing the general bug-fixing Java data shared in [32] as their fine-tuning dataset. We utilized the bug-fixing Java data [32] as the training data and followed Wu et al. to examine the correctness of the generated patches by using the available test cases and manual verification.

Results. Considering that the recent work [75] utilized large model sizes for their baselines (up to 12 billion parameters), we introduced a larger variant of VulMaster in this discussion to align with the baselines. We use CodeT5p-large (0.8 billion parameters) as the backbone model in this experiment. The results, presented in Table 7, clearly demonstrate that VulMaster achieved the highest performance. Out of 35 vulnerabilities, VulMaster repaired 9, achieving a fix rate of 25.7%.

6.4 Explanations for Low Fix Rate

VulMaster achieves a relatively low fix rate, primarily due to the inherent complexity of certain vulnerabilities. To delve into the causes of these incorrect fixes, we conducted a manual analysis. We randomly sampled 100 vulnerable-fixed code pairs from the test set and examined the corresponding generated fixes. Out of the 100 sampled vulnerabilities, VulMaster produced 78 incorrect fixes. Our manual investigation of these 78 failures revealed major causes as follows: 1) Vulnerabilities requiring multiple-hunk code changes pose a significant challenge for VulMaster. This category accounted for 31 failures. Generating accurate multi-hunk patches remains a challenging task for learning-based models. Notably, many state-of-the-art APR techniques [33, 34, 78, 86] do not support bugs that require multi-hunk code changes. 2) For 13 failures, developers' correct patches incorporate members of structure/union variables that are not defined within the vulnerable functions, which means those members are unknown to VulMaster. Thus, it is unlikely for VulMaster to generate the correct fixes. 3) For 8 failures, VulMaster fails to produce identical strings as the developers' correct patches. 4) For 5 failures, VulMaster fails to generate complex conditions in the correct fixes. 5) For 5 failures, VulMaster fails to use the user-defined APIs correctly. 6) For 4 failures, VulMaster uses the wrong APIs. 7) For 5 failures, VulMaster generates semantically equivalent but syntactically different patches. Since we use Exact Match as the evaluation metric, these patches are considered 'incorrect' fixes. Besides, 7 other failures cannot be classified.

6.5 Threats to Validity

Threats to internal validity pertain to potential errors and biases in our experiments. To address these threats, we take several precautions. Firstly, we strictly follow the settings and methodologies employed by baselines, using their official implementations to ensure consistency. We have reviewed and validated our code and data, which are publicly accessible for transparency and reproducibility. Threats to external validity relate to the generalizability of VulMaster. To mitigate this potential concern, we meticulously select the experimental dataset, metrics, and baselines. The vulnerability repair dataset comprises 1,754 open-source software projects and has been utilized in prior learning-based state-of-the-art approaches. For the metrics, we select three widely used metrics, including the EM, BLEU, and CodeBLEU. Existing work [14] has proven the reliability of these metrics. Lastly, we follow prior works [9, 19, 75] and assume perfect fault localization. We acknowledged that obtaining perfect or near-perfect localization is challenging. We consider more effective fault localization to be beyond the scope of this paper. We consider a human-in-the-loop setting where an experienced software engineer sifts through the output of a fault localization tool and only employs our approach once the location of the vulnerability is identified. We encourage future work to continue looking into better methods to localize faults, extending the popular line of work on fault localization.

7 RELATED WORK

Vulnerability Datasets. Previous research has introduced various datasets to facilitate the evaluation of vulnerability repair approaches. The ManyBugs [40] dataset consists of 185 defects in 9 open-source programs. Most of their defects are logical errors rather than security vulnerabilities. The VulnLoc [61] dataset contains 43 vulnerable programs from 10 projects that span 6 CWEs. The Vul4J [7] dataset includes reproducible vulnerabilities from 51 open-source Java projects, representing 25 different CWEs. CVEFixes [5] and Big-Vul [15] are two large vulnerability repair datasets without test cases, curated from 1,754 projects. The large number of projects in CVEFixes and Big-Vul datasets help ensure the diversity of vulnerabilities stored in them.

Learning-based Vulnerability Repair. A number of learning-based approaches for repairing vulnerabilities have been proposed by researchers, extending the line of work on learning-based program repair, e.g. [39]. For example, Ma et al. [47] proposed Vurle, the first learning-based approach for vulnerability repair; it learns transformative edits and their contexts from examples of vulnerable code fragments and their repairs. Chi et al. [10] introduced SeqTrans, a Transformer-based machine translation model with copy mechanisms designed to fix Java vulnerabilities. Chen et al. [9] proposed VRepair, which initially pre-trains a vanilla Transformer model on a bug-fixing corpus and subsequently utilizes it to address C/C++ vulnerabilities. Fu et al. [19] proposed VulRepair, which leverages a BPE tokenizer and CodeT5 that is pre-trained on a large code corpus. Wu et al. [75] conducted an evaluation of nine large language models and four program repair models on the Vul4J dataset [7]. In contrast, VulMaster is specifically designed to effectively utilize diverse input sources: 1) input vulnerable functions, 2) code structures, and 3) rich CWE expert knowledge, while leveraging the power of Large Language Models. Our goal is to advance the progress of learning-based vulnerability repair approaches.

Program Analysis-based Vulnerability Repair. Several program analysis-based approaches for repairing vulnerabilities have been proposed. CDRep [46], one of the earliest program analysis-based vulnerability repair approaches, employs a set of templates and data flow analysis to fix cryptographic-related vulnerabilities in Android apps, with a success rate of 90%. SenX [29] aims to repair vulnerabilities by leveraging vulnerability-specific and human-specified safety properties. SAVER [27] and Memfix [41] are designed to address memory errors. FootPatch [66] generates patches that adhere to specific heap properties specified using separation logic. FootPatch is constrained to addressing only a limited set of common vulnerability types, such as memory leaks. VulnFix [80] utilizes counterexample-guided inductive inference for repairing vulnerabilities. However, VulnFix is not applicable to vulnerabilities that cannot be resolved by modifying or inserting conditions, or situations that necessitate the introduction of new program variables [80]. Unlike CDRep, SenX, SAVER, Memfix, FootPatch, and VulnFix, VulMaster is not limited to specific vulnerability types. Fix2Fit [20] uses fuzz testing to filter out patches (produced by the APR model) that cause crashes. Combining VulMaster and Fix2Fit has the potential to enhance overall performance. CPR [60] employs concolic execution, along with a user-provided specification, to generate new inputs that aid in identifying and filtering out overfitting patches among the generated ones. ExtractFix [21] uses dependency analysis and symbolic executions to fix vulnerabilities. In contrast to CPR and ExtractFix, VulMaster does not depend on heavy concolic and symbolic executions, making it able to generate patches more efficiently.

VulMaster differentiates itself from program analysis-based vulnerability repair approaches in several key aspects. Firstly, as a learning-based approach, VulMaster is not limited to a specific type of vulnerability, distinguishing it from various program analysis-based methods like SenX [29], SAVER [27], Memfix [41], FootPatch [66], and VulnFix [80]. However, for vulnerability types where good examples are hard to find, and specialized algorithms, precise templates, or formal safety properties are available, program analysis-based solutions will perform better. Thus, the two lines of work (learning-based and program analysis-based vulnerability repair) are complementary. Secondly, VulMaster is adaptable to multiple programming languages, different from many program analysis-based vulnerability repair methods (such as SAVER [27], Fix2Fit [20], and ExtractFix [21]) focusing solely on C/C++. Thirdly, VulMaster can generate patches more efficiently (about 10-20 seconds for one vulnerability) than program analysis-based vulnerability repair methods relying on heavy symbolic and concolic executions (e.g., CPR [60] and ExtractFix [21]) or time-consuming dynamic invariant inference (e.g., VulnFix [80]).

8 CONCLUSION AND FUTURE WORK

We propose VulMaster, a vulnerable repair model that effectively mitigates the input length limitations of Transformer-based pre-trained models, following FiD. By eliminating the length limit, VulMaster can integrate diverse information, including vulnerable code structures and expert knowledge from the CWE system, to further enhance its repair capabilities. The experimental results demonstrate that VulMaster outperforms all baselines significantly. A replication package is provided at **https://github.com/soarsmu/VulMaster_**.

In the future, our interests will extend across multiple directions. As there is potential redundancy or noise in the inputs, we plan to design a refinement and denoising module to mitigate these issues. Moreover, we are keen on employing larger pre-trained code models for this task in the future, utilizing parameter-efficient tuning techniques like [73]. Additionally, we plan to develop *trustworthy and synergistic* AI4SE methodologies [45] to help software engineers better trust and synergize with VulMaster and other automated vulnerability identification and repair solutions.

Acknowledgement. This research / project is supported by the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). 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.

REFERENCES

[1] 2016. The example commit details. https://github.com/TinkerBoard2-Android/external-ImageMagick/commit/dd84447b63a71fa8c3f47071b09454efc667767b.

[2] 2023. Replication Package of VulMaster. https://github.com/soarsmu/VulMaster_.

[3] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. 2023. Recommending Root-Cause and Mitigation Steps for Cloud Incidents using Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1737–1749. https://doi.org/10.1109/ICSE48619.2023.00149

[4] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 143–153.

[5] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[7] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E Díaz Ferreyra. 2022. Vul4J: a dataset of reproducible Java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 464–468.

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

[9] 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.

[10] 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.

[11] CWE Community. 2023. Common Weakness Enumeration: CWE. https://cwe.mitre.org/.

[12] CWE Community. 2023. Homepage of CWE-401. https://cwe.mitre.org/data/definitions/401.html.

[13] CVE Community. 2023. Official website of Common Vulnerabilities and Exposures. https://www.cve.org/.

[14] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software* 203 (2023), 111741.

[15] 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.

[16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[17] Nicole Forsgren, Bas Alberts, Kevin Backhouse, Grey Baker, Greg Cecarelli, Derek Jedamski, Scot Kelly, and Clair Sullivan. 2021. 2020 state of the octoverse: Securing the world's software. *arXiv preprint arXiv:2110.10246* (2021).

[18] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2020. VulRepair Official Repository. https://github.com/awsm-research/VulRepair.

[19] 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.

[20] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 8–18.

[21] 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.

[22] GitHub. 2021. The 2020 state of the octoverse. https://octoverse.github.com/.

[23] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, and M. Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. *ArXiv* abs/2009.08366 (2021).

[24] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. 2009. *The elements of statistical learning: data mining, inference, and prediction*. Vol. 2. Springer.

[25] Junda He, Xin Zhou, Bowen Xu, Ting Zhang, Kisub Kim, Zhou Yang, Ferdian Thung, Ivana Clairine Irsan, and David Lo. 2023. Representation Learning for Stack Overflow Posts: How Far are We? *ACM Transactions on Software Engineering and Methodology* (2023).

[26] Hossein Homaei and Hamid Reza Shahriari. 2017. Seven years of software vulnerabilities: The ebb and flow. *IEEE Security & Privacy* 15, 1 (2017), 58–65.

[27] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: scalable, precise, and safe memory-error repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 271–283.

[28] 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.

[29] 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.

[30] Gautier Izacard and Edouard Grave. 2021. Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*. Association for Computational Linguistics, Online, 874–880. https://doi.org/10.18653/v1/2021.eacl-main.74

[31] Tiantian Ji, Yue Wu, Chang Wang, Xi Zhang, and Zhongru Wang. 2018. The coming era of alphahacking?: A survey of automatic software vulnerability detection, exploitation and patching techniques. In *2018 IEEE third international conference on data science in cyberspace (DSC)*. IEEE, 53–60.

[32] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (2023), 1430–1442. https://api.semanticscholar.org/CorpusID:256808267

[33] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair. In *Proceedings of the International Conference on Software Engineering*.

[34] 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.

[35] D. Jurafsky and J. H. Martin. 2014. Speech and language processing, 3rd ed. *Hoboken, New Jersey: PrenticeHall, Inc.* (2014).

[36] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.

[37] Linux Kernel. 2019. The Example Commit from Linux Kernel. https://github.com/CGCL-codes/SCVDT/blob/0a30f2e59f45168407cb520e16479d53d7a8845b/VulnDB/FuncSamples/linux_kernel/CVE-2019-19073/CVE-2019-19073_CWE-400_853acf7caf10b828102d92d05b5c101666a6142b_htc_hst.c_1.1_htc_connect_service_OLD.vul.

[38] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[39] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 213–224.

[40] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.

[41] 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.

[42] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[43] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. 2022. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1035–1047.

[44] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. 2022. A unified multi-task learning model for AST-level and token-level code completion. *Empirical Software Engineering* (Jul 2022). https://doi.org/10.1007/s10664-022-10140-7

[45] David Lo. 2023. Trustworthy and Synergistic Artificial Intelligence for Software Engineering: Vision and Roadmaps. *CoRR* abs/2309.04142 (2023). https://doi.org/10.48550/ARXIV.2309.04142 arXiv:2309.04142

[46] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. 2016. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS*

2016, Xi'an, China, May 30 - June 3, 2016, Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang (Eds.). ACM, 711–722. https://doi.org/10.1145/2897845.2897896

[47] 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 (Lecture Notes in Computer Science, Vol. 10493)*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer, 229–246. https://doi.org/10.1007/978-3-319-66399-9_13

[48] 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.

[49] Yao Meng and Long Liu. 2020. A deep learning approach for a source code detection model using self-attention. *Complexity* 2020 (2020), 1–15.

[50] Jonathan Mirault, Joshua Snell, and Jonathan Grainger. 2018. You that read wrong again! A transposed-word effect in grammaticality judgments. *Psychological Science* 29, 12 (2018), 1922–1929.

[51] Van Nguyen, Trung Le, Tue Le, Khanh Nguyen, Olivier DeVel, Paul Montague, Lizhen Qu, and Dinh Phung. 2019. Deep domain adaptation for vulnerable code function identification. In *2019 international joint conference on neural networks (IJCNN)*. IEEE, 1–8.

[52] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[53] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An empirical comparison of pre-trained models of source code. In *2023 IEEE/ACM 45rd International Conference on Software Engineering (ICSE)*.

[54] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. Spt-code: Sequence-to-sequence pre-training for learning source code representations. In *Proceedings of the 44th International Conference on Software Engineering*. 2006–2018.

[55] OpenAI. 2023. Chatgpt official blog. https://openai.com/blog/chatgpt.

[56] OpenAI. 2023. GPT homepage. https://openai.com/research/gpt-4.

[57] 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.

[58] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[59] Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal V. Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Févry, Jason Alan Fries, Ryan Teehan, Teven Le Scao, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M. Rush. 2022. Multitask Prompted Training Enables Zero-Shot Task Generalization. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. https://openreview.net/forum?id=9Vrb9D0WI4

[60] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic program repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 390–405.

[61] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. 2021. Localizing vulnerabilities statistically from one exploit. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 537–549.

[62] Tiezhu Sun, Kevin Allix, Kisub Kim, Xin Zhou, Dongsun Kim, David Lo, Tegawendé F Bissyandé, and Jacques Klein. 2023. Dexbert: Effective, task-agnostic and fine-grained representation learning of android bytecode. *IEEE Transactions on Software Engineering* (2023).

[63] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, et al. 2023. Automatic Code Summarization via ChatGPT: How Far Are We? *arXiv preprint arXiv:2305.12865* (2023).

[64] ED TARGETT. 2022. We analysed 90,000+ software vulnerabilities: Here's what we learned. https://www.thestack.technology/analysis-of-cves-in-2022-software-vulnerabilities-cwes-most-dangerous/.

[65] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2022. Efficient Transformers: A Survey. *ACM Comput. Surv.* 55, 6, Article 109 (dec 2022), 28 pages. https://doi.org/10.1145/3530811

[66] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*. 151–162.

[67] Cunxiang Wang, Haofei Yu, and Yue Zhang. 2023. RFiD: Towards Rational Fusion-in-Decoder for Open-Domain Question Answering. *arXiv preprint arXiv:2305.17041* (2023).

[68] Deze Wang, Yue Yu, Shanshan Li, Wei Dong, Ji Wang, and Liao Qing. 2021. MulCode: A Multi-task Learning Approach for Source Code Understanding. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. https://doi.org/10.1109/saner50967.2021.00014

[69] Xin Wang, Xiao Liu, Pingyi Zhou, Qixia Liu, Jin Liu, Hao Wu, and Xiaohui Cui. 2022. Test-Driven Multi-Task Learning with Functionally Equivalent Code Transformation for Neural Code Generation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. https://doi.org/10.1145/3551349.3559549

[70] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[71] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *CoRR* abs/2109.00859 (2021). arXiv:2109.00859 https://arxiv.org/abs/2109.00859

[72] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5 Model Release from Huggingface. https://huggingface.co/Salesforce/codet5-base.

[73] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2023. Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models. *arXiv preprint arXiv:2308.10462* (2023).

[74] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.

[75] 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 *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[76] 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*.

[77] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

[78] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 1506–1518.

[79] Aiping Zhang, Liming Fang, Chunpeng Ge, Piji Li, and Zhe Liu. 2023. Efficient transformer with code token learner for code clone detection. *Journal of Systems and Software* 197 (2023), 111557.

[80] 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.

[81] Xin Zhou, DongGyun Han, and David Lo. 2021. Assessing generalizability of codebert. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 425–436.

[82] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, Junda He, and David Lo. 2023. Generation-based Code Review Automation: How Far Are We ?. In *31st IEEE/ACM International Conference on Program Comprehension, ICPC 2023, Melbourne, Australia, May 15-16, 2023*. IEEE, 215–226. https://doi.org/10.1109/ICPC58990.2023.00036

[83] Xin Zhou, Kisub Kim, Bowen Xu, Jiakun Liu, DongGyun Han, and David Lo. 2023. The Devil is in the Tails: How Long-Tailed Code Distributions Impact Large Language Models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 40–52.

[84] Xin Zhou, Bowen Xu, DongGyun Han, Zhou Yang, Junda He, and David Lo. 2023. CCBERT: Self-Supervised Code Change Representation Learning. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 182–193.

[85] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).

[86] 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.