T-RAP: A Template-guided Retrieval-Augmented Vulnerability Patch Generation Approach

Pei Liu
liupei@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Bo Lin
linbo19@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Yihao Qin
yihaoqin@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Cheng Weng
wengcheng@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Liqian Chen*
lqchen@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

ABSTRACT

Vulnerabilities exert great burden on developers in terms of debugging and maintenance. Automated Vulnerability Repair(AVR) is considered as a promising approach to alleviate the burden of developers. Template-based automated program repair techniques have shown their effectiveness in fixing general bugs. However, due to the diverse root causes of vulnerabilities, it is challenging to construct sufficient repair templates to cover various vulnerabilities. In this paper, we introduce a Template-guided Retrieval-Augmented Patch generation approach, named T-RAP. Inspired by retrieval-augmented techniques that effectively utilize historical data, our approach leverages repair templates to extract similar vulnerability repair patches from the codebase. These patches then guide the process of generating vulnerability patches. To extract similar patches, we also propose a matching algorithm specifically designed for the retrieval-augmented vulnerability repair. This involves identifying similarities between numerous templates and vulnerabilities during the template-guided stage. Experimental results demonstrate that T-RAP outperforms all the studied AVR approaches, repairing 56.8% more vulnerabilities than VulRepair and 30.24% more than VulMaster. It can also accurately repair more types of real-world vulnerabilities than VulMaster. Additionally, we evaluated the effectiveness of our patch retriever. The results indicate that our template-guided retriever, which is based on our matching algorithm, outperforms the retrieval algorithm proposed in the recent retrieval-augmented patch generation approach RAP-Gen.

*Liqian Chen is the corresponding author.

Internetware 2024, July 24–26, 2024, Macau, China

ACM ISBN 979-8-4007-0705-6/24/07
https://doi.org/10.1145/3671016.3672506

CCS CONCEPTS

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

KEYWORDS

Software Vulnerability, Automated Vulnerability Repair, Repair Template, Deep Learning

ACM Reference Format:
Pei Liu, Bo Lin, Yihao Qin, Cheng Weng, and Liqian Chen. 2024. T-RAP: A Template-guided Retrieval-Augmented Vulnerability Patch Generation Approach. In *15th Asia-Pacific Symposium on Internetware (Internetware 2024), July 24–26, 2024, Macau, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3671016.3672506

1 INTRODUCTION

Over the years, owing to the increasing quantity and complexity of vulnerabilities, software has become more susceptible to potential attacks. Traditional approaches to fixing vulnerabilities heavily rely on the manual efforts of developers. The manual identification and resolution of these bugs demand significant investments of time and resources [1].

To alleviate the arduous burden of manual fixing activities [13], Automated Vulnerability Repair (AVR) techniques have emerged as a promising way to automatically correct vulnerabilities. Generally speaking, AVR techniques consist of three major stages: vulnerability localization, patch generation and patch validation. Among these stages, the quality of the patches generated by repair procedures directly determines the repair results and is crucial to the entire repair process. Consequently, lots of existing AVR techniques are dedicated to generating more accurate patches [6, 7, 12].

Template-based AVR techniques are a prominent group of AVR techniques, which utilize predefined program fix templates to generate repair patches [16, 25, 33]. Such predefined templates are often manually summarized by researchers, which is rather time-consuming and limited greatly by the domain knowledge of designers. Also, existing template-based AVR techniques mainly focus on type-specific vulnerabilities repair, such as buffer overflow [16] and

integer overflow [25], because they have similar root causes and thus are easy to define templates manually. However, as of February 2024, the total number of vulnerability categories in the Common Weakness Enumeration (CWE) has reached 938 [9], indicating the difficulty in summarizing templates for each type of vulnerability manually, especially considering that many types of vulnerabilities lack explicit fix templates (e.g., CWE-1001: Use of an Improper API) [3]. In addition, existing template-based automated repair approaches are not truly automated and usually time-consuming. Although templates auto-mining techniques have reduced the manual efforts in summarizing templates and solved the problem of a limited number of manually defined templates, these templates can not be applied on the vulnerabilities directly. Designing and implementing associated repair editions remains a heavy burden for developers. Therefore, existing template-based automated repair approaches are limited in type-specific vulnerability repair and are not fully automatic.

Meanwhile, alongside the flourishing development of deep learning technology [20], some researchers leverage the power of language models to repair vulnerabilities. Specifically, learning-based AVR techniques regard the vulnerability repair task as a code-to-code translation task that learns repair actions from historical data. Consequently, these techniques rely heavily on the quantity and quality of historical vulnerability repair data, which can be hardly satisfied by existing vulnerability datasets [8]. The current mainstream public vulnerability datasets Big-Vul [11] and CVEFixes [2] consist of 3,754 and 5,465 pairs of vulnerabilities respectively, covering 91 and 180 CWE types. Limited datasets pose challenges for learning-based vulnerability repair solutions, making the optimization of existing datasets crucial for AVR techniques.

Aiming to make full use of existing vulnerability datasets, retrieving relevant code snippets and building connections between similar data may help generating desired code, which can reduce search space and provide essential code in the code generation process [30]. This is consistent with human mind, that is, when faced with an vulnerability to be repaired, experienced developers will first try to search relevant code snippets for reference in other large-scale codebases [28, 29], such as Github. Developers accumulate a wealth of knowledge and experience throughout the development and maintenance phases, often producing various documents to categorize encountered problems in team collaboration. This enables them to readily access similar code snippets or information to assist in their repair tasks. However, for automated repair techniques, the challenge lies in establishing connections between vulnerabilities and their corresponding code snippets for reference.

To mitigate the aforementioned limitations observed in both template-based and learning-based AVR techniques, and inspired by human developers' retrieving ways for fixing vulnerabilities, we propose T-RAP, a Template-guided Retrieval-Augmented Patch generation approach. T-RAP begins by automatically extracting repair templates, which not specifically targeted at vulnerability bugs, but cover general repair actions derived from a group of historical bug-fixing pairs. The repair pairs are then categorized into different groups according to the corresponding templates. When given a piece of vulnerable code that needs fixing, T-RAP searches for the adopted repair templates at the Abstract Syntax Tree (AST) level by our matching algorithm (detailed in Section 3.1.2). This stage yields relevant vul-fix pairs based on the matched templates. Subsequently, T-RAP retrieves several similar vul-fix pairs from those recorded in the matched templates, which serve as references for repairing the given vulnerable code. We fine-tune a code-aware language model CodeT5 as our patch generator. In the final stage, the patch generator uses both the vulnerable code and the retrieved bug-fixing pairs as inputs to generate a candidate patch.

We evaluate the effectiveness of T-RAP on the dataset BigFixes (which we construct by merging two existing vulnerability datasets Big-Vul [11] and CVEFixes [2]). The results show that T-RAP outperforms all studied AVR approaches, repairing 30.24% more vulnerabilities than state-of-the-art AVR technique VulMaster.

In summary, the contributions of this paper are as follows:

- We propose a template-guided retrieval-augmented patch generation method, leveraging repair templates which are high-level abstractions of a set of similar repair actions, to retrieve relevant vul-fix pairs from the codebase, and then use the retrieved repair pairs to guide the patch generation.
- We propose a template matching algorithm designed for the matching task between numerous templates mined automatically and multiple suspicious statements. This algorithm involves categorizing templates and assigning priorities to facilitate the identification of the optimal match.
- We implement our approach by constructing a new tool called T-RAP. We thoroughly evaluate T-RAP on BigFixes dataset. Experimental results show the efficiency of our repair procedure. When compared to existing vulnerability repair tools such as VulRepair and VulMaster, we can generate more accurate patches. We also prove the usefulness of our retrieval-augmented method by contrast with other retrieval algorithms.

2 BACKGROUND

2.1 Automated Vulnerability Repair

AVR is derived from Automated Program Repair (APR), which is an integrated and automated process of vulnerability repair. The main process framework of AVR task includes three modules: (1) localization module to identify the vulnerability code location; (2) patch generation module to automatically modify the vulnerability code and generate patches to eliminate the root cause of vulnerability; (3) patch verification module to check the correctness of the candidate patches [21].

In order to better understand the root causes and repair mechanisms of various types of vulnerabilities, researchers categories vulnerabilities and form specific vulnerability types known as CWEs. According to whether they target specific types of vulnerabilities, existing AVR methods can be divided into type-specific and generic vulnerability repair methods. Type-specific AVR methods design the localization and patch generation algorithm based on the syntactic and semantic features of specific types of vulnerabilities, with the advantage of high precision. However, with the increasing number of vulnerability categories in CWE system, it is challenging to design repair techniques for all types of vulnerabilities. Therefore, it is promising to research on the generic AVR methods. Generic AVR methods usually take the repair task as a translation process, which aims to translate the buggy program into the correct program by leveraging various Neural Machine Translation (NMT)

approaches [6, 7, 24]. Existing AVR techniques like VRepair and SeqTrans, both adopt the transfer learning strategy by pre-training on bug fixing datasets and fine-tune on vulnerability datasets. VRepair has obtained a repair accuracy of 17.3%. Other generic repair method VulRepair adopt a pre-trained encoder-decoder model based on the T5 architecture and obtained a repair accuracy of 44%.

2.2 Retrieval-Augmented Generation

Retrieval-augmented generation has achieved state-of-the-art performance in many NLP tasks [26]. Retrieval source, retrieval metric and integration methods are three major components of the retrieval-augmented generation paradigm. Most studies fetch the retrieval source from its training corpus, that is, in the generation inference phase, they retrieve relevant examples from the training corpus. Retrieval metrics determine which examples in the retrieval source are relevant to the input sequence. BM25 [34] is the widely used sparse-vector retrieval methods, which both compute similarity at the lexical level. After obtaining the retrieval information, data augmentation is the one of the most straightforward way to integrate the retrieved into the generation process, which constructs the augmented inputs by concatenating the retrievals with the original input. There are many applications of retrieval-augmented generation in different generation tasks, such as code generation [15] and machine translation [4, 14].

Wang *et al.* introduced a retrieval-augmented generation approach to automated program repair tasks, called RAP-Gen [31]. RAP-Gen constructs a hybrid patch retriever to find a relevant bug-fix pair first, augmenting the original buggy input, and then train on the code-aware language model CodeT5 model to generate candidate patches. RAP-Gen demonstrates superior performance over state-of-the-art deep learning-based methods on three benchmarks in JavaScript and Java, indicating the promising potential of retrieval-augmented generation in advancing APR tasks.

3 APPROACH

In this section, we introduce our approach, T-RAP, a template-guided retrieval-augmented patch generation method, leveraging repair templates to retrieve relevant vul-fix pairs from the codebase, and then use the retrieved repair pairs to guide the patch generation. The workflow of T-RAP is illustrated in Fig. 2. Our T-RAP architecture comprises three primary phases: 1) the template-indexing retriever construction; 2) the patch generator training with pre-trained model; 3) the inference phase for patch prediction.

3.1 Templates-indexing Retriever

T-RAP aims to use templates as indexes to find the relevant historical code changes, and then use the retrieved reference to augment the patch generation. To that end, our workflow begins with constructing a template-indexing retriever, whose retrieving outputs are important reference ingredients for instructing the next patch generation phase. Specifically, we initially extract repair templates from historical vul-fix commits and associate these templates with the corresponding vul-fix pairs (i.e., the commits from which the templates were extracted). Subsequently, for a given vulnerability V_t, T-RAP selects the best-matching template with the corresponding relevant vul-fix pairs $\langle V_i, P_i \rangle$ to serve as the reference for repairing V_t (illustrated in Sec 3.1.2).

3.1.1 Templates mining. We adapt an automated technique for mining vulnerability repair templates by leveraging FixMiner [17] specifically tailored for vulnerabilities in the C/C++ language. Initially, we transform all code alterations into Rich Edit Scripts (RES) [17], which utilize four actions (i.e., update, insert, delete, and move) and node types in AST to represent the code modifications as an edit script [10]. The RES preserves contextual details of code changes, such as the AST structure and the altered node type, thereby providing richer information for template matching. Subsequently, based on the collected RESs, we identify clusters of trees with identical structures. From these clusters, we derive a set of RESs sharing the same structure, constituting a template when the cluster contains at least two members. Each template is an abstraction of several code changes in the same cluster, and encapsulates identical repair modifications applied to the same types of nodes. To facilitate the future use, we establish a dictionary to store the mapping between templates and the group of relevant vul-fix pairs, with the template serving as the key and the group of vul-fix pairs as the value.

```
--- UPD if
------ UPD condition
--------- UPD expr
------------ UPD call
--------------- UPD name
------------ DEL operator
```

(a) Template

```
- if(PyUni_CompWithASCII(name, *p) == 0) {
+ if(_PyUni_EqualToASCII(name, *p)) {

- if(PyUni_CompWithASCII(name, "_debug_") == 0){
+ if(_PyUni_EqualToASCII(name, "_debug_")) {
```

(b) Relevant vul-fix diff code

Figure 1: One of templates that related to if_stmt and relevant vul-fix diff code.

Fig. 1 shows an example of a template along with two of the relevant vul-fix pairs represented in the GNU diff format. It can be observed that we have mined a template that modifies the expression within the condition of an if statement, and both the two vul-fix diff patches in Fig. 1(b) involve deleting the operator and updating the call function within the condition. Our miner captures this characteristic and extracts common edits on the nodes of the same types, forming a template as Fig. 1(a). Leveraging the dictionary we built, we can directly retrieve the two vul-fix pairs shown in Fig. 1(b) by the template in Fig. 1(a).

3.1.2 Templates matching. After obtaining a set of templates and building the dictionary from templates to the corresponding vul-fix pairs within clusters in the template-mining phase, we intend to construct our template-guided retriever next. As for an vulnerable

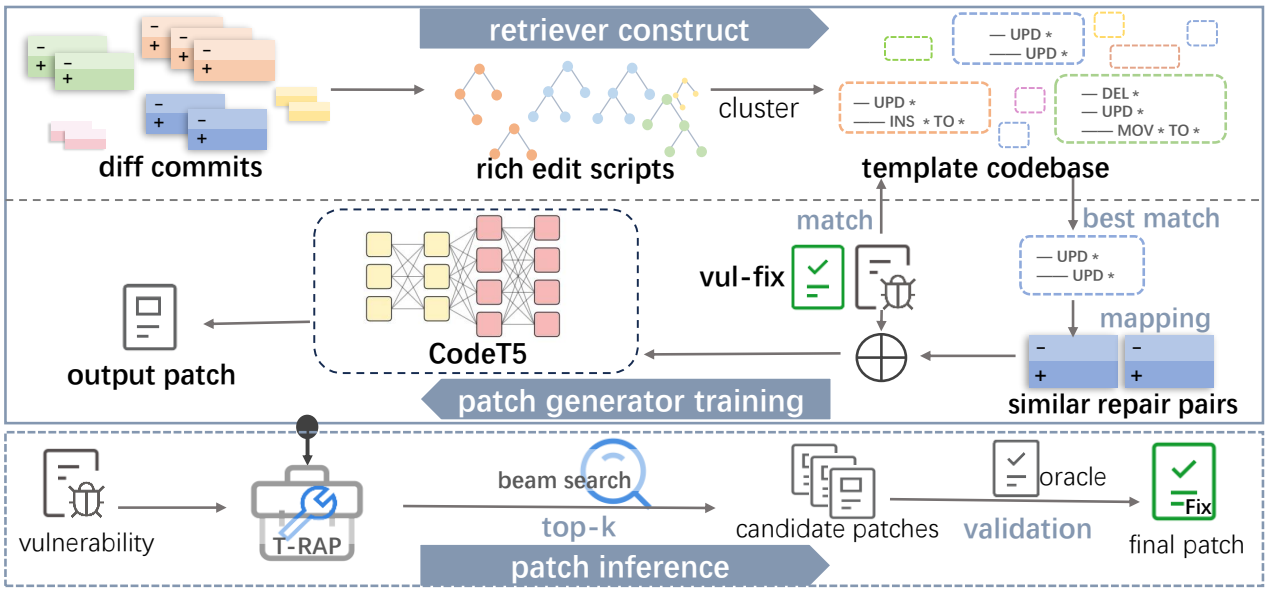

Figure 2: The overall workflow of T-RAP.

code snippet V_t to be repaired, when we want to construct a retriever to search code changes similar to this snippet for reference like developers, template can be a good bridge to build connections between the two parts. Therefore, we convert the task of building the mapping between vulnerable code to the similar historical code changes in the codebase, to the task of selecting the most suitable fix template for the vulnerable code snippet. The template matching algorithm are implemented in in Algorithm 1.

To match the code snippet and templates mined, we leverage a popular AST parser, tree-sitter[1], to convert the code into an abstract tree, and then extract the buggy nodes at the labelled location lines, namely from `<StartBug>` to `<EndBug>`, which are key nodes in the next matching. Traditional template-based APR techniques [22] manually predefine the bug context for each template. They traverse each node of the suspicious statement from its first child node to the last leaf node. If a node matches the predefined bug context, the corresponding template is selected for patch generation. This template selection process continues until a correct patch is generated. Considering that there may be multiple suspicious lines labelled in an vulnerable code snippet and usually not only one buggy node at each labelled line, and the number of templates mined by FixMiner is in the hundreds, it is infeasible to iteratively traverse all templates one by one in the codebase to match with the context of the vulnerable code V_t until a correct patch is validated.

Therefore, we devise a matching algorithm tailored for the situation of matching between several buggy nodes and numerous templates mined automatically in AVR task, aiming to select the best match template. This initiative draws upon the foundational matching strategy prevalent in traditional template-based APR techniques [22, 23], which matches the buggy code context with templates in the AST level. However, we introduce several refinements into this strategy to enhance its applicability to our template-guided approach. First, according to the modified node type, we organize the templates mined in the template codebase into different type groups, including `if_stmt`, `return`, `expr_stmt`, `decl_stmt`, `return`, `for`, `while`, `function` and `block_content`. We note that the modified node types exhibit varying levels of granularity and distinctive features. For instance, modifications within the `expr_stmt` templates and `block_content` templates are obviously not at the same granularity, with the `expr_stmt` also serving as a constituent of flow control statements templates like `if_stmt`. The presence of overlapping characteristics among the templates may result in an vulnerable code snippet matching successfully with several templates, thereby obscuring the realization of best match. For example, Fig. 3(a)is an vulnerable code snippet with fault location labels, with one of its buggy nodes `(type = identifier, "realloc")`. As for the context of this buggy node in the AST, several templates can be matched successfully, such as Fig. 3(b) and Fig. 3(c). We expect to select template in 3(c) as the best match, for it reflect the if control structure in the vulnerable function 3(a).

Consequently, we categorize these templates into three broader groups: (1) flow control statement (if_stmt, for, while); (2) basic statement (expr_stmt, decl_stmt, return); (3) high granularity block (function, block_content). The priority order is defined the same as above, with flow control statements assigned the highest priority, because we think the flow control statements are more important label as the context information for matching. We assign templates featuring high granularity block the lowest priority, for we expect to make fewer modifications in repair.

After selected the template group, we conduct the matching process as the Algorithm 1. The matching algorithm begins by iterating over each template in the templates_group. For each template, the algorithm initializes two variables: best_match to track the highest

[1] https://tree-sitter.github.io/tree-sitter/

```
 if (bitmapUpdate->number > bitmapUpdate->count){
    ...
    /*<StartBug>*/
    newdata = (BITMAP_DATA *)realloc(bitmapUpdate->rectangles,\
    /*<EndBug>*/  sizeof(BITMAP_DATA) * count);
    ...
}
```

(a) An example of vul buggy node

```
---UPD expr_stmt
------ UPD expr
--------- UPD call
------------ UPD name
```

(b) Template expr_stmt

```
--- UPD if
------ UPD block
------------ DEL expr_stmt
--------------- DEL expr
------------------ DEL call
---------------------- DEL name
---------------------- DEL argument_list
```

(c) Template if_stmt

Figure 3: An example of vulnerable snippets and the matched templates.

match score encountered so far, and best_template to store the corresponding template. Within the loop, we initialize a temporary score variable score and then invoke the node_match function to compare the attributes of the buggy_node with the current template. This function recursively traverses both the buggy node and the template, comparing their attributes to determine the degree of similarity. The node_match function takes as input the buggy node (b), the current template node (t), the depth of traversal in both the buggy node (Lb) and the template node (Lt), and a flag (all_traversed) indicating whether all nodes at the same depth in the buggy context have been completed. If the traversal encounters a leaf node in the template or if all_traversed is True, the function will return the current depth of the template (Lt). Otherwise, it proceeds to compare the types and attributes of the current nodes (b.type and t.type). If they match, the function recursively explores their child nodes to further assess the similarity with the depth incremented both. If no match is found, the function backtracks and explores alternative paths. Once the traversal is complete, the function returns the depth of the template node (Lt) where the best match was found. Back in the match function, if the score obtained from node_match is greater than or equal to the current best_match, the best_template is updated to the current template.

3.1.3 Templates indexing. With the best match template prepared, we can access a set of relevant historical code changes by the dictionary built during templates mining. Given that the difference between vulnerable and the repaired code typically occur in contiguous lines of code [23], which contains the key information of the patch. Therefore, we extract the modified statements along with the surrounding lines of code from both the vulnerable code and fixed code, to form the retrieved vul-fix pairs.

Algorithm 1 T-RAP template-matching algorithm

Input: *buggy_node*, *templates_group*
Output: *best_template*

```
function MATCH(buggy_node, templates_group)
    for each template in templates_group do
        best_match ← 0
        best_template ← ϕ
        T ← jsonTree(template)
        score =NODE_MATCH(buggy_node, T, 0, 0, false)
        if score ≥ best_match then
            best_template ← template
        end if
    end for
    return best_template
end function
function NODE_MATCH(b, t, Lb, Lt, all_traversed)
    if !T.children or all_traversed is true then
        return Lt
    end if
    if b.type == t.type then
        if !T.children then
            return Lt
        else
            for each b_c in b.children do
                for each t_c in t.children do
                    Lb, Lt ← Lb + 1, Lt + 1
                    Lt = NODE_MATCH(b_c, t_c, Lb, Lt, false)
                end for
                all_traversed ← True
            end for
        end if
    end if
end function
```

3.2 Data Processing

3.2.1 Output representation. Early works [12, 27] on neural repair has proved that outputting the whole function resulted in decreased performance as the function length increased. We follow the code representation designed in previous data-driven vulnerability repair methods [6], which only outputs the changes on source code tokens, instead of the whole generated function with the same length as source code. We use a pair of special tokens <StartBug> and <EndBug> to label the suspicious location code, which guide the further modifying location for patch generation. As for the code with multiple suspicious locations, label each one respectively, and allow for generating multiple changes in the output. Meanwhile, we use another pairs of special tokens <StartMod> and <EndMod> to label the start and end of the changes in the source code. Combining the source code and the output patch identifying the specific fix could yield the entire function with fix changes. Figure4 shows an example of our output representation of the vul-fix pairs with labeled tokens, and T-RAP aims to generate correct patches in the representation of the fix in the example.

3.2.2 Subword Tokenization. We adopt the Byte Pair Encoding (BPE) algorithm as our tokenizer, consistent with prior researches [12,

```
" vul ":  ... 1 ] ; <StartBug> assert ( 0 ) ; <EndBug>  ...
" fix ":        <ModStart> 1 ] ; return NULL ; <ModEnd>
```

Figure 4: An example of vul-fix pair representation.

31, 32]. BPE operates by tokenizing words at the byte-level and effectively reducing the size of the vocabulary, consequently diminishing the dimensions of the embedding layer. Besides, the BPE algorithm mitigates the Out-of-Vocabulary (OOV) issue. OOV refers to words or symbols encountered during model training that are not included in the vocabulary. The occurrence of OOV instances can greatly affect the model's performance and its ability to generalize, as it cannot understand words outside its vocabulary, resulting in incorrect interpretation of input data.

3.3 Patch Generation

The generator of T-RAP adopts an encoder-decoder architecture, comprising 12 Transformer layers in both the encoder and decoder. Each layer utilizes 12 attention heads for multi-head attention computations, resulting in a total parameter size of 220M. This architecture is widely employed in state-of-the-art pre-trained models and across various application scenarios [5, 12, 18, 31, 32]. Consistent with prior researches [12, 18, 31], we initialize the parameters of each generator with weights from CodeT5 [32]. The purpose of this initialization is to provide the generator with domain-specific knowledge of programming languages, which can help T-RAP generate more reliable patches.

3.3.1 Input Preparation. Given an vulnerability pairs $\langle V_t, F_t \rangle$ from the training dataset, namely function V_t and its correct fix F_t, the template-indexing retriever outputs several relevant vul-fix code changes$\langle V_i, F_i \rangle$,$\langle V_j, F_j \rangle$... if there are templates matched successfully. In patch generator training phase, we design the strategy that all of the retrieved vul-fix pairs, along with the V_t, are used as input to the patch generator to generate the predicted patch. We use a retrieval field R to denote the retrieved code change pairs $\langle V_i, F_i \rangle$,$\langle V_j, F_j \rangle$... , separating different pairs with a special token $[SEP]$. That is: $R_t = \langle V_i, F_i \rangle [SEP] \langle V_j, F_j \rangle$.... As for the given the training data pair $\langle V_t, F_t \rangle$, the input to our patch generator is augmented as : $I_t = V_t \oplus R_t$.

There are also vulnerabilities fail to match any of the templates mined, indicating that no similar historical repairs can be retrieved in the codebase for reference. On this occasion, the retrieval filed R will be NULL, and the inputs to our patch generator will include the vulnerable code only, that is: $I_t = V_t$.

3.3.2 Model Training. After integrating the retrieval field with the original vulnerable input, we prepare the augmented training data to patch generator. This is denoted as : $I_t = V_t[RETR]V_i[SEP]F_i$..., where [RETR] is a special token separating the retrieved vul-fix pairs from the original vulnerable code, and [SEP] is a special token separating the vul-fix pairs. Specifically, the training input pairs$\langle I_t, F_t \rangle$ will first be tokenized by BPE into a sequence of code at the sub-word level, then the Embedding layer generate an embedding vector for each subword, capturing the semantic features of the code tokens. The results of word embedding serves as the input to the encoder module of T-RAP, deriving the encoder hidden state, which in turn informs the decoder module to generate and output the predicted patches. The primary objective of model training is to fine-tune the model to obtain specific weights for repair patch generation task on the vulnerability dataset. Therefore, we adopt cross-entropy loss function over the training phase, which is frequently used in optimization and probability estimation. Formally, the loss can be described as:

$$\mathcal{L}_{oss} = -\sum_{i=1}^{N} \log(P_\theta(F_t | I, F_{<t})).$$

where N denote the number of tokens in the patch, θ are the parameters of model, I is the ground truth, and $F_{<t}$ is the tokens generated so far.

3.4 Patch Inference

During inference, we employ the beam search strategy to generate a list of candidate fixed patches for a given vulnerable function. Specifically, as for the given vulnerable function V_d, (1) We use Tree-sitter, a widely-used AST parser generator, to construct the AST and extract a list of its faulty nodes. Each faulty code is then matched with the mined templates (illustrated in 3.1.2). We combine the vector V_d with the retrieved code changes, which are mapped from the matched template. (2) We then split V_d using the tokenization component and input the subword tokenized sequence into the trained model. (3) The model extracts and decodes the sequence's features, then generates the probability distribution of the tokens that constitute the candidate patch. We implement beam search to select several candidate patches based on the probability at each decoding step. That is, the beam size parameter β is determined to select the Top-β high probability predicted patches generated during the encoding step. These candidate patches are later validated for their correctness.

4 EXPERIMENTAL DESIGN

4.1 Research Questions

We seek to answer the following research questions (RQs):

RQ1: Effectiveness of T-RAP. How does T-RAP perform to repair vulnerabilities in open source projects compared with other AVR approaches?

RQ2: Analysis of T-RAP predictions. How does T-RAP perform in fixing vulnerabilities of different types and complexity?

RQ3: Ablation Study. What are the contributions of the major components of T-RAP?

4.2 Dataset

To evaluate the T-RAP more accurately, we constructed a dataset named BigFixes by combining the two largest vulnerability datasets, Big-Vul [11] and CVE-Fixes [2]. We conducted a comparison of the vulnerability source files and eliminated any duplicates in the merged dataset. This was done to prevent data leakage, ensuring that the same item does not appear in both the training and testing sets. As a result, we obtained a consolidated dataset with 6,008 vulnerabilities, covering 180 CWE types. We adopt the same data

spilt way as Chen *et al.* [6] and Fu *et al.* [12], allocating 70% of the dataset is for training, 10% for validation and 20% for testing.

4.3 Baselines

We choose the most advanced and relevant AVR techniques VRepair and VulRepair as the baseline to compare the repair performance with our approach T-RAP.

VRepair. Chen *et al.* [6] proposed a Transformer-based vulnerability repair approach VRepair in 2021. VRepair adopted transfer learning technique, which first trained on a large bug fix corpus and fine-tuned on the vulnerability dataset in C, alleviating the small vulnerability dataset problem.

VulRepair. Fu *et al.* [12] proposed a T5-based vulnerability repair approach named VRepair in 2022. VulRepair adopted CodeT5 model and fine-tuned the model on the vulnerability repair task, and achieved a better repair results compared to VRepair.

VulMaster. Zhou et al. [35] proposed VulMaster, VulMaster effectively understands vulnerable code, regardless of its length, and integrates diverse information, including code structures and expert knowledge from the CWE system. VulMaster achieves better repair results compared to VRepair and VulRepair.

RAP-Gen. RAP-Gen[31] adopts a hybrid patch retriever for bug-fix pairs mining, which considers both lexical and semantic matching by combining sparse and dense retrievers. We implemented the retriever of RAP-Gen based on the design outlined in the paper, which is not open-sourced.

T-RAP-BM25 and **T-RAP-Rand**. To evaluate the contribution of our template-guided retriever on the repair performance, we propose two contrasting methodologies with different retrieving strategies: the T-RAP-BM25 method with BM25 algorithm realizing the retrieval, and the T-RAP-Rand method utilizing a random retriever. Both methodologies leverage CodeT5 as patch generator.

4.4 Evaluation Metrics

Different from benchmarks in APR field, there are no test cases provided in existing vulnerability datasets. Existing AVR techniques [6, 12] are devoting to generating patches that can be exactly matched with the oracle fix, that is, a candidate patch is considered correct only if it is exactly matched with the oracle fix. Therefore, T-RAP validate the generated patches by exactly matching with the oracle fix following existing automated repair techniques [6, 12, 31, 35].

We utilize Recall to evaluate the repair performance of T-RAP, which is widely used in automatic repair evaluation metric. Recall is the percentage of the number of vulnerabilities fixed by the correct patches generated in all the vulnerabilities. The higher recall means the higher quality of the patches generated by the repair techniques. We compare the recall of our T-RAP with other baseline approaches in accuracy metrics on our BigFixes.

In addition, considering that it is time-consuming in validating the correctness of patches generated by beam search, we also adopt the metric Recall@Top-k, which measures the percentage of the correct patch being in the top-k predicted patches generated of all the vulnerabilities, and the smaller k means the patches generated are more precise and a better repair performance. In our experiments, we evaluate the accuracy of patches generated at the Top-1, Top-5 and Top-10 respectively.

4.5 Implementation Details

T-RAP is mainly composed of two modules, which is the template-guided retriever and the retrieval-augmented patch generator.

FixMiner [17] is a template mining tool based on an iterative clustering strategy to produce fix templates for APR systems, which selects projects written in Java with bug reports as dataset, and focuses on evaluating the relevance of the yielded templates [17]. In our template-guided retriever, we transfer FixMiner into focusing on vulnerabilities functions in C language and evaluating the effectiveness of our template-guided retriever. We first extract repair templates on our training dataset to construct our template database, and then represent the repair templates mined in text as json trees with keys of node_type, edit_operation and children_nodes, so that we can match vulnerabilities and templates based on tree structures. After matching vulnerabilities with templates as our matching algorithm1, we store the relevant vul-fix pairs retrieved by the matched templates in a new field noted as "retrievals". which will be the input of the patch generator training phase together with the oracle vul-fix pairs later.

T-RAP construct the patch generator based on a T5 model with an encode-decode architecture in each generator, which is widely used by state-of-the-art pre-trained models[5, 12, 18, 19, 31, 32]. We initialize the parameters of each generator with the weights from CodeT5-base [32] to equip the generators with domain-specific knowledge of programming languages, and then fine-tune these generators for the downstream task of patch generation. We run the experiments on a server with 2 Nvidia GeForce RTX 4090 GPUs. We fine-tune the model for 30 epochs with the learning rate 3e-4 and batch size 4, using the AdamW optimizer to update the model and minimize the loss function. In the inference phase, we set the beamsize as 50, the same with the baseline technique VulRepair.

5 EXPERIMENTAL RESULTS

5.1 RQ1: How does T-RAP perform to repair vulnerabilities in open source projects compared with other AVR approaches?

To evaluate the performance of T-RAP in open source projects, we compare T-RAP with other four baseline approaches on BigFixes dataset. Table 1 shows the effectiveness of T-RAP compared to other state-of-the-art AVR techniques. Generally speaking, T-RAP outperforms these AVR techniques in repair recall metrics. Specifically, the recall of T-RAP is 146.6% and 56.8% higher than VRepair and VulRepair, 30.24% and 14.9% higher than VulMaster and RAP-Gen, which means T-RAP is capable of generating more correct patches for vulnerabilities. We also concentrate on the top-1 accuracy, which determines the cost of human effort due to the lack of test cases. We find that T-RAP can generate 28.90% and 18.13% more correct patches than VulMaster and RAP-Gen at the top-1 candidate patch rank, which illustrates T-RAP can achieve a good repair performance with less patches generated and validated.

Comparison Analysis. The recall of VulMaster, RAP-Gen and T-RAP all surpass 20%. Compared with VRepair and VulRepair, these techniques all leverage additional information beyond the original vul-fix pairs. VulMaster has integrated diverse information from

Table 1: Comparison with state-of-the-art AVR techniques.

	Recall	@Top-1	@Top-5	@Top-10
VRepair	10.90%	6.82%	10.15%	10.57%
VulRepair	17.14%	13.31%	16.06%	16.72%
VulMaster	20.63%	17.23%	18.72%	20.08%
RAP-Gen	23.38%	18.80%	22.55%	22.90%
T-RAP	26.87%	22.21%	25.04%	25.80%

the CWE system and expert knowledge, therefore succeed in generating the most unique patches. The superior repair performance of RAP-Gen and T-RAP shows the effectiveness of retrieval-augmented strategy in generating patches. Retrievals establish the internal connections among repair commits, maximizing the utilization of limited datasets during model training. Templates-guided retriever of T-RAP outperforms than hybrid retriever of RAP-Gen. This maybe because repair templates can capture deeper connections inherent in repair actions behind repair commits, while hybrid retrieve may be more inclined to focus on the code attributes themselves.

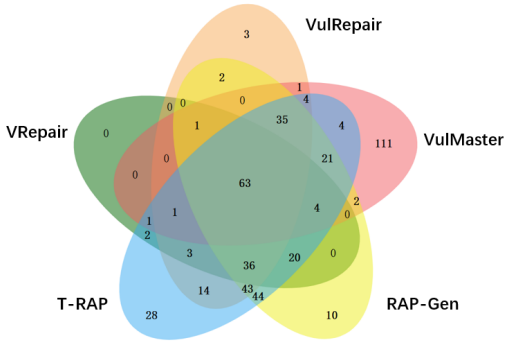

Figure 5: Overlaps of vulnerabilities fixed by different techniques.

Overlap Analysis. To further investigate how well T-RAP complements VRepair, VulRepair, VulMaster and RAP-Gen, we analyse the number of overlapping vulnerabilities fixed by the above techniques. As is shown in Fig 5, T-RAP fixes 28 more unique vulnerabilities than existing four techniques, and generated 75 and 42 more correct patches than VulMaster and RAP-Gen respectively. Overall, the results illustrate that T-RAP is complementary to the four state-of-the-art AVR techniques to increase the number of vulnerabilities being correctly fixed in BigFixes.

5.2 RQ2: How does T-RAP perform in fixing vulnerabilities of different types and complexity?

To better analyse the repair performance of T-RAP in patch generation, we perform investigations in the CWE types and the lengths of the vulnerabilities that can be repaired by T-RAP.

We analyzed the CWEs with the highest frequency in our test dataset and calculate the Recall metrics respectively. Table 2 shows the repair performance of T-RAP on vulnerabilities of different CWE types. It can be seen that T-RAP achieves a balanced performance in the recall of different CWE types of vulnerabilities, ranging from 23% to 36%. T-RAP achieves the best performance on CWE-190 and CWE-264, reaching 36.84% and 36.00%, while obtains the worst performance on CWE-20 and CWE-416 with 23.81% both.

Table 2: The repair performance of T-RAP on vulnerabilities of different CWE types.

CWE Type	CWE Descriptions	Recall	Proportion
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	26.30%	81/308
CWE-125	Out-of-bounds Read	25.00%	18/112
CWE-20	Improper Input Validation	23.81%	25/105
CWE-264	Permissions, Privileges, and Access Controls	36.00%	18/50
CWE-476	NULL Pointer Dereference	25.00%	11/44
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	20.93%	9/43
CWE-416	Use After Free	23.81%	10/42
CWE-190	Integer Overflow or Wraparound	36.84%	14/38
CWE-787	Out-of-bounds Write	28.12%	9/32
CWE-399	Resource Management Error	35.48%	11/31
TOTAL		26.83%	216/805

In addition, we counted the CWE types of vulnerabilities fixed by different AVR techniques. As Fig 6 shows, T-RAP also repairs 3 more CWE types of vulnerabilities than VulMaster, which makes fully use of the rich expert knowledge beyond the CWE type from the CWE system and is good at tackling vulnerabilites with varieties of CWE type. We also note that T-RAP repaired 2 unique CWE types(CWE-269 and CWE-674), which have not been repaired by the above four techniques. T-RAP failed in generating correct patches for vulnerabilities of CWE-863, CWE-672 and CWE-276, because the number of these types of vulnerabilities in BigFixes is less than 10, hardly to support retrieving and patch generating.

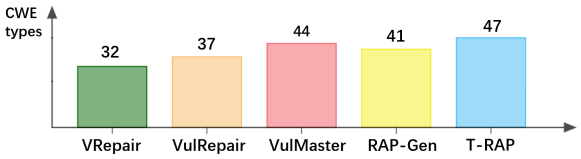

Figure 6: CWE types of vulnerabilities fixed by different techniques.

To investigate the effect of different lengths of vulnerabilities on the repair performance of T-RAP, we divide the our dataset into several ranges with interval 100 according to the token length after subword tokenization, and analyse the recall metric on different length intervals respectively. The result is as Table 3. We find that the performance of T-RAP decreases as the length increases on the whole. T-RAP achieves the best performance on the vulnerabilities on the tokens length less than 100 with 50.52% accuracy, while the recall substantially drops to 25.41% for vulnerable functions with tokens between 300 to 400. Considering the T5 model's maximum input length of 512 tokens, any additional tokens beyond this threshold will be truncated, leading to a sharp decrease in recall for vulnerabilities exceeding 500 tokens. Furthermore, vulnerabilities longer than 500 tokens constitute more than half of our dataset, thereby contributing to the overall low repair accuracy of our T-RAP.

Table 3: The repair performance of T-RAP on vulnerabilities of different length intervals.

Length Interval	Recall	proportion
(0,100)	50.52%	49/97
(101,200)	36.90%	62/168
(201,300)	33.11%	49/148
(301,400)	25.41%	31/122
(401,500)	33.33%	36/108
>500	17.17%	96/559

5.3 RQ3: What are the contributions of the major components of T-RAP?

To investigate the impact of the template-guided retriever in our T-RAP approach, we excluded our template-retriever from T-RAP for ablation experiments. In this training phase, the input to the patch generator was the original vulnerable function along with the oracle fix without retrievals. We set another two groups of experiments: T-RAP-Rand and T-RAP-BM25 to further explore the effects of different retrievers.

The results are presented in Table 4. It is worth noting that the recall decreased 34.8% after removing our template-retriever, which proves the effectiveness of our template-guided retriever in repair performance. Our template-guided retriever, utilized by T-RAP, repaired 50.1% and 21.8% more vulnerabilities compared to T-RAP-Rand and T-RAP-BM25 respectively. We observed that the repair accuracy of T-RAP-Rand closely aligns with the recall of VulRepair, primarily due to its adaptation of CodeT5 in the patch generation process.

Table 4: Results of Ablation Study.

Ablation Settings	Recall
T-RAP without template-retriever	17.50%
T-RAP-Rand	18.97%
T-RAP-BM25	22.05%
T-RAP	26.87%

6 RELATED WORK

Template-based AVR. Existing template-based AVR techniques are mainly aimed at type-specific vulnerabilities, for they are the same in the root cause thus easy to define templates manually. Existing research on template-based AVR mainly focus on common specific types of vulnerabilities such as buffer overflow and interger overflow, and more feasible repair techniques need to be proposed for a large number of other types of vulnerabilities. In addition, existing template-based automated repair approaches are not truly automated, in which templates can not be applied on the vulnerabilities directly.

Different from existing template-based AVR techniques, T-RAP does not restrict the types of vulnerabilities. Templates in T-RAP plays a role of guiding the repair, not the skeleton of generating repair patches, therefore can fit in a variety types of vulnerabilities and can be directly applied on the patch generation.

Learning-based AVR. Chen *et al.* [6] proposed a Transformer-based vulnerability repair approach VRepair. VRepair adopted transfer learning technique, which first trained on a large bug fix corpus and fine-tuned on the vulnerability dataset in C. Fu *et al.* [12] proposed a T5-based vulnerability repair approach VulRepair. VulRepair adopt Code-T5 pre-trained language model and fine-tuned the model on the vulnerability repair task.

Different from existing learning-based AVR techniques, inspired by the way of developers in repairing, T-RAP retrieves historical repairs to augment the inputs in the patch generation.

7 DISCUSSION

7.1 Compared with Large Language Models

To assess the effectiveness of Large Language Models (LLMs) in automated vulnerability repair, we use two renowned LLMs: the GPT-3.5-turbo-0125 model and DeepSeek-Coder. The input consists of the vulnerable function and the vul-fix pair retrieved by our template-guided retriever. The output follows the same settings. The prompt is: "You are an automated vulnerability repair technique. Here are some historical fixes for reference: . . . Please return the repaired modified code for the following vulnerable function: . . .". With the temperature set to zero, the LLMs always return the most likely repair. Thus, we only utilize the Top-1 recall as the metric. The results show that the GPT-3.5-turbo-0125 and DeepSeek-Coder can repair 6.1% and 4.9% of vulnerabilities, respectively. This could be because current advanced methods are fully fine-tuned on task-specific datasets. In contrast, ChatGPT is optimized for artificial general intelligence, not specifically for bug fixes.

7.2 Threats to Validity

Internal Validity. The threats to internal validity mainly comes from the following three aspects: 1) the retrieval results by templates. For we retrieve historical repair commits by matching the context of the buggy nodes in the AST level, there may be irrelevant repair commits retrieved and interfere the model training, thereby result in the incorrect patch generated; 2) the parsing errors by tree-sitter. There may be parsing errors if a new version of a programming language introduces new syntactic features or syntactic rules, and the current version of Tree-sitter has not been updated to support these changes; 3) the hyperparameter settings of T-RAP. Different hyperparameter settings will lead to different evaluation results in neural models, but it is expensive and time-consuming to find an optimal hyperparameter setting. Therefore, we do conduct a grid search to find a hyperparameter setting with a relatively better performance , but cannot make sure current setting is the best.

External Validity. Our approach is only evaluated on the vulnerability dataset in C language and covering 180 different CWEs. We do not study its generalization to the vulnerability datasets of other programming languages (PLs) and other CWEs. Our approach is language-specific for we employ the features like ASTs in the template-mining module, and thus cannot be generalized to other PLs directly. We leave equipping our approach being language-agnostic as our future work.

Construct Validity We rely on exact match accuracy to evaluate the repair performance following previous studies [6, 12, 31]. EM can reflect the repair performance at a syntactic level, but do not

consider the semantics of patches. That is, patches semantically equivalent but not exactly match with the oracle fix will not be considered as a correct patch. However, limited by the absence of test cases in existing vulnerability datasets, other metrics cannot be imported into AVR techniques. We expect to see new vulnerability datasets equipped with test cases where repairs can be reproducible in the future work.

8 CONCLUSION

In this paper, we propose a template-guided retrieval-augmented patch generation approach (T-RAP) for automated vulnerability repair. T-RAP consists of two components: a template-guided retriever to retrieve the relevant historical repair pairs, and a retrieval-augmented patch generator to output the patches with the vulnerable function and retrievals as input. Through the evaluation of the repair accuracy and the analysis of the vulnerabilities fixed, we show that our T-RAP achieves the best performance in repair accuracy than the four studied AVR techniques. All data in this study are publicly available at: **https://zenodo.org/records/11001126**.

REFERENCES

[1] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2020. On the effectiveness of unified debugging: An extensive study on 16 program repair systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 907–918.

[2] 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.

[3] Quang-Cuong Bui, Ranindya Paramitha, Duc-Ly Vu, Fabio Massacci, and Riccardo Scandariato. 2024. APR4Vul: an empirical study of automatic program repair techniques on real-world Java vulnerabilities. *Empirical software engineering* 29, 1 (2024), 18.

[4] Deng Cai, Yan Wang, Huayang Li, Wai Lam, and Lemao Liu. 2021. Neural Machine Translation with Monolingual Translation Memory. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, Online, 7307–7318.

[5] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. 2022. NatGen: Generative Pre-Training by "naturalizing" Source Code *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 18–30.

[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] Roland Croft, M Ali Babar, and M Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 121–133.

[9] CWE. 2024. Common Weakness Enumeration. Website. https://cwe.mitre.org.

[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM, 313–324.

[11] 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.

[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* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 935–947.

[13] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67.

[14] Jiatao Gu, Yong Wang, Kyunghyun Cho, and Victor O.K. Li. 2018. Search engine guided neural machine translation *(AAAI'18/IAAI'18/EAAI'18)*. AAAI Press, Article 629, 8 pages.

[15] Tatsunori B. Hashimoto, Kelvin Guu, Yonatan Oren, and Percy Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 10073–10083.

[16] 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.

[17] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* 25, 3 (2020), 1980–2024.

[18] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 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 (ESEC/FSE 2022)*. 1035–1047.

[19] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. 2023. CCT5: A Code-Change-Oriented Pre-trained Model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1509–1521.

[20] Bo Lin, Shangwen Wang, Zhongxin Liu, Xin Xia, and Xiaoguang Mao. 2022. Predictive Comment Updating with Heuristics and AST-Path-Based Neural Learning: A Two-Phase Approach. *IEEE Transactions on Software Engineering* (06 2022).

[21] Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. 2021. Context-Aware Code Change Embedding for Better Patch Correctness Assessment. *ACM Transactions on Software Engineering and Methodology* (12 2021), 1.

[22] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 31–42.

[23] 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.

[24] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: learning to repair compilation errors. In *Proceedings of the 2019 27th ACM ESEC/FSE*. 925–936.

[25] Paul Muntean, Martin Monperrus, Hao Sun, Jens Grossklags, and Claudia Eckert. 2021. IntRepair: Informed Repairing of Integer Overflows. *IEEE Transactions on Software Engineering* 47, 10 (2021), 2225–2241.

[26] S. M Towhidul Islam Tonmoy, S M Mehedi Zaman, Vinija Jain, Anku Rani, Vipula Rawte, Aman Chadha, and Amitava Das. 2024. A Comprehensive Survey of Hallucination Mitigation Techniques in Large Language Models.

[27] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28 (2018), 1 – 29.

[28] Shangwen Wang, Mingyang Geng, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Li Li, Tegawendé Bissyandé, and Xiaoguang Mao. 2023. Natural Language to Code: How Far Are We?

[29] Shangwen Wang, Mingyang Geng, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Li Li, Tegawendé Bissyandé, and Xiaoguang Mao. 2024. Fusing Code Searchers. *IEEE Transactions on Software Engineering* PP (01 2024), 1–15.

[30] Shangwen Wang, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Yan Lei, and Xiaoguang Mao. 2023. Two Birds with One Stone: Boosting Code Generation and Code Search via a Generative Adversarial Network. *Proceedings of the ACM on Programming Languages* 7.

[31] 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 FSE*.

[32] 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.

[33] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: value-flow-guided precise program repair for null pointer dereferences. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 512–523.

[34] Stephen E. Robertson;Hugo Zaragoza. 2009. The probabilistic relevance framework: BM25 and beyond(Article). *Foundations and Trends in Information Retrieval* (2009), 333–389. Issue No.4.

[35] 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 *International Conference on Software Engineering*.