RATCHET: Retrieval Augmented Transformer for Program Repair

Jian Wang[1,2], Shangqing Liu[2†], Xiaofei Xie[1], Jingkai Siow[2], Kui Liu[3] and Yi Li[2]
[1]Singapore Management University, Singapore
[2]Nanyang Technological University, Singapore
[3]Software Engineering Application Technology Laboratory, Huawei, China

***Abstract*—Automated Program Repair (APR) presents the promising momentum of releasing developers from the burden of manual debugging tasks by automatically fixing bugs in various ways. Recent advances in deep learning inspire many works in employing deep learning techniques to fixing buggy programs. However, several challenges remain unaddressed: (1) state-of-the-art fault localization techniques often require additional artifacts, such as bug-triggering test cases or bug reports. These artifacts are not always available in the early development phases; (2) Sequence-to-Sequence model-based APR often requires additional contexts with high quality to generate patches. Yet, it is challenging to identify high-quality contexts that are not common in programs.**

In this paper, with the redundancy assumption in program repair, we propose a dual deep learning-based APR tool, RATCHET, for localizing (RATCHET-FL) and repairing (RATCHET-PG) buggy programs. RATCHET-FL localizes buggy statements based on the feature learned by a simple BiLSTM model from the code, without any bug-triggering test cases or bug reports. RATCHET-PG relies on our proposed retrieval augmented transformer to learn the historical patches and generate patches for fixing bugs. We evaluate the effectiveness of RATCHET with in-the-lab DrRepair dataset and in-the-wild dataset RATCHET-DS (curated in this work). Our experimental results show that RATCHET outperforms state-of-the-art deep learning approaches on fault localization with 39.8-96.4% accuracy and patch generation with 18.4-46.4% repair accuracy.

I. INTRODUCTION

Debugging is a routine task in software development and maintenance, taking significant time and efforts. Especially with the increasing number of open-source libraries and the expanding size of software systems, the number of software defects has been increasing rapidly [1]. However, these defects are commonly introduced by software developers with no intention. As reported by LaToza et al. [2], developers spent 50% of their time in discovering and fixing bugs. Furthermore, in contrast to the quantity of open source projects and the speed of software iteration, the number of exploited defects is far from enough, i.e., existing a large number of "silent" defects, which have not been exploited [1]. Hence, researchers have been exploring automated fault localization and program repair techniques to release developers from the heavy burden of manual debugging, but it still is far from a settled problem.

Automated program repair (APR) mainly consists of a two-step process: Localization (localizing the bug) and Auto-Patch (generating the patches for the localized bugs). In the literature, most APR works [3]–[8] employ spectrum-based fault localization (SBFL) techniques to expose the bug positions and fix the located bugs through a generate-and-validate approach. Popular SBFL techniques, such as Ochiai [9] and Tarantula [10], localize the bug positions by leveraging the execution traces of negative and positive test cases. These techniques compute a suspicious score based on the frequency of each statement in full execution traces of all test suites. One major limitation of the SBFL techniques is that they highly rely on the bug-triggering test case(s) (i.e., the failed executed test case(s)). Once the buggy statement is localized, patches are automatically generated in three ways: ① the heuristic-based approaches [4], [6]–[8], that construct and iterate over a search space of syntactic program modifications with dedicated mutate operators; ② the constraint-based approaches [11]–[13] that build on constraint solving to synthesize transformations for patch generation; and ③ learning-based approaches [14]–[17] which explore machine learning techniques to boosting program repair by learning correct code or natural transformation. Heuristic-based approaches suffer from search space explosion (i.e., enumerating more patch candidates) as the search space of the patches will be widely enlarged due to the increasing amount of mutation operators [4], [18], while constraint-based approaches require execution paths to generate constraints, of which compilation and validation processes take up a huge amount of time and resources [18], and inadequate constraints increase the possibility of generating plausible patches [19]. Recently, learning-based approaches [3], [14]–[17] have presented outstanding performance on fixing bugs by generating fewer patch candidates with fewer time costs when comparing against the state-of-the-art heuristic-based and constraint-based program repair approaches.

The learning-based approaches [3], [14]–[17], [20]–[22] for the program repair (e.g., the Seq2Seq model [15]) take as input the buggy function [15], [20] or buggy statements [16], [17], [21] for the encoder, and decode the extracted features applied to the patch generation. To contribute to enhancing the quality of the generated patches, various contexts have been exploited as the feature supplier of patch generation. CoCoNut [16] and CuRe [17] engages the surrounding lines of a bug as its contexts and insert them into a context encoder for contextual learning. SequenceR [15] considers the buggy function as the context of the bug and feeds it into the

†Corresponding author.

Seq2Seq model for the patch inference. Such context-based learning approaches indeed achieved promising performance on fixing bugs, nevertheless, they simply leverage the LSTM-based model to encode the local contextual information for the learning process. Incorporating contexts for bug-related feature learning could introduce noises and hinder the learning process. As stated by Tufano et al. [20] and Chen et al. [15], generating fixes for the completed buggy function can result in lower accuracy.

In addition, fault localization is the first step of automated program repair, but CoCoNut [16], SequenceR [15], and CuRe [17] are evaluated with the perfect fault localization assumption [23] that the buggy statement can be accurately localized by a perfect fault localization tool, which however totally ignores the impact from the fault localization. DrRepair [14] considers the compiler messages to localize error and generate more accurate patches, which however cannot provide any hints for localizing functional bugs that deviate from developers' intention. DLFix [3] requires a larger number of test cases with SBFL techniques to proceed with the fault localization. The state-of-the-art SBFL techniques [24] present more practical performance on exposing bug positions than other techniques, but a large number of bugs still cannot be accurately localized by them [23] and they highly rely on the bug-trigger test cases that are always unavailable in the real bug cases [25]. Nevertheless, the accuracy of fault localization could impact the quality of patches generated by APR tools [19].

The redundancy assumption of program repair [26], [27] points out that the buggy code is recurrent in the big codebase and the related patches can be grafted for similar bugs. Indeed, code duplication [28] is expected in the "big code" era and retrieving similar code has already proved the effectiveness in many DL4Code tasks (e.g., commit message generation [29], [30] and code summarization [31]). Based on the redundancy assumption, we propose combining code retrieval with the advanced transformer to boost fault localization and patch generation for program repair.

In this work, we propose a dual deep learning-based program repair tool, RATCHET[1], with two different deep learning models: RATCHET Fault Localization model (RATCHET-FL) and RATCHET Patch Generation model (RATCHET-PG). RATCHET-FL formulates the localization as a classification problem to predict the buggy statement via Bi-Directional Long-Short Term Memory network, without any bug-triggering test cases or bug reports. With the localized buggy statement, RATCHET-PG employs our proposed retrieval augmented transformer to generate patches for it. Specifically, inspired from the recent work [31] on the improvement of the retrieval information, we propose to incorporate buggy statements with the closest retrieved patches retrieved from the historical patches for them via the retrieval-augmented layer in the transformer to contribute the patch generation. This paper makes the following contributions:

- RATCHET-DS, a curated bug-patch pair dataset with 56,974 cases collected from 13 popular open-sourced C/C++ projects, of which bugs and patches are collected systematically. Dataset RATCHET-DS and SourceCode are publicly available at https://sites.google.com/view/apr-ratchet.
- RATCHET, a dual deep learning-based program repair tool by integrating RATCHET-FL with RATCHET-PG. RATCHET outperforms state-of-the-art learning-based APR tools on fixing both in-the-lab bugs with 19.5% repair accuracy and the in-the-wild bugs with 46.4% repair accuracy. And, RATCHET is not so sensitive to the fault localization setting for generating correct patches.
- RATCHET-FL, a BiLSTM fault localization model without using bug-triggering test cases or bug reports. Experimental results show that RATCHET-FL can effectively identify buggy statements in the given buggy functions, which outperforms the state-of-the-art retrieval/learning-based approaches with the Acc@Top1, Acc@Top3, and Acc@Top5 metrics at 39.8-96.4% for two datasets.
- RATCHET-PG, a novel Retrieval Augmented Transformer model with the retrieval-augmented layer to integrate the closest retrieved patches with the buggy statements to generate correct patches. With the closest retrieved patches, RATCHET-PG can generate correct patches for more (16-236) bugs.

II. USAGE SCENARIO AND PROBLEM FORMULATION

In this section, we describe the usage scenario of RATCHET and formalize its fault localization and patch generation task.

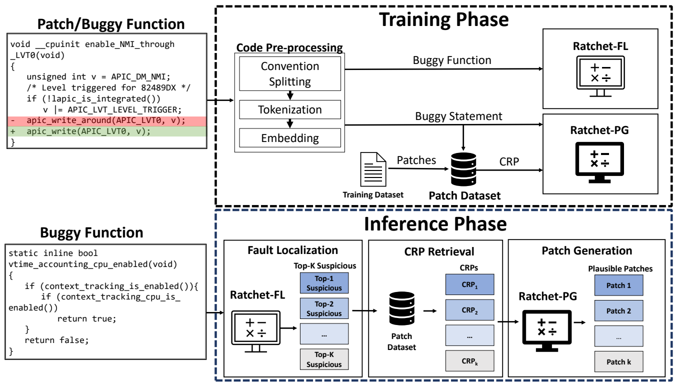
A. Usage Scenario

RATCHET incorporates the fault localization and patch generation into a unified framework. It treats the fault localization as a multi-classification problem to predict the bug position with a well-trained LSTM model, which is different from the state-of-the-art program repair tools that rely on the existing fault localization techniques [3]–[5], [12], [20], [21] or build on the perfect fault localization assumption [3], [15]–[17]. RATCHET takes as input the buggy function to predict the exact buggy statement. The buggy function is accessible to off-the-shelf fault localization tools [32], [33]. To simplify the complexity of program repair, followed by other auto-patching works [3], [15]–[17], [21], RATCHET focuses on the buggy functions with the single buggy statement and generates the patches for them.

B. Problem Formulation

To simplify the framework of RATCHET, we formalize two sequential tasks: fault localization (RATCHET-FL) and patch generation (RATCHET-PG). For RATCHET-FL, given a buggy function c where $c = \{s_1, s_2, ..., s_n\}$ and n is the total lines of c, we aim to find the buggy statement s_l via predicting the line number l by localization function f_{loc}, $l = f_{loc}(c)$, $l \in [1, n]$. For RATCHET-PG, when buggy statement s_l is localized, the generation function f_{gen} is used to produce the patch statement

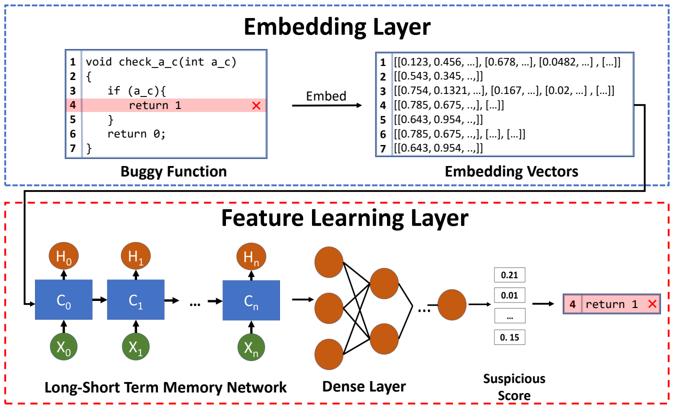
[1]Ratchet is a Chief Medical Officer of Autobot. It fixes robots, which is a metaphor for our tool-repairing programs.



```
void __cpuinit enable_NMI_through
_LVT0(void)
{
   unsigned int v = APIC_DM_NMI;
   /* Level triggered for 82489DX */
   if (!lapic_is_integrated())
      v |= APIC_LVT_LEVEL_TRIGGER;
-  apic_write_around(APIC_LVT0, v);
+  apic_write(APIC_LVT0, v);
}
```

```
static inline bool
vtime_accounting_cpu_enabled(void)
{
   if (context_tracking_is_enabled()){
      if (context_tracking_cpu_is_
enabled())
         return true;
   }
   return false;
}
```

Fig. 1: Overview of RATCHET.



```
1 void check_a_c(int a_c)
2 {
3     if (a_c){
4         return 1
5     }
6     return 0;
7 }
```

```
1 [[0.123, 0.456, ...], [0.678, ...], [0.0482, ...] , [...]]
2 [[0.543, 0.345, ..,]]
3 [[0.754, 0.1321, ...], [0.167, ...], [0.02, ...] , [...]]
4 [[0.785, 0.675, ..,], [...]]
5 [[0.643, 0.954, ..,]]
6 [[0.785, 0.675, ..,], [...], [...]]
7 [[0.643, 0.954, ..,]]
```

Fig. 2: Architecture of RATCHET-FL model.

$s_p = f_{gen}(s_l)$. We tackle the two tasks by devising two neural networks to approximate f_{loc} and f_{gen} for the solving.

III. Approach

The overall framework of RATCHET is illustrated in Fig. 1, which consists of three phases: ① Training ana RATCHET-FL model, which is applied to localize which statement is faulty; ② Training an RATCHET-PG model, which is used to generate the patch candidates for the faulty statement; ③ Inference, when a buggy function is given, RATCHET first uses RATCHET-FL to localize the bug position and uses RATCHET-PG to generate the patch.

As illustrated by Fig. 1, both the fault localization model RATCHET-FL and the patch generation model RATCHET-PG of RATCHET are first well-trained with the buggy code and patched code from collected human-written patches. Given a dataset $D = \{f = (c, l, p) | c \in C, l \in L_c, p \in P_c\}$ where c is the buggy function in the buggy-function set C, RATCHET first leverages its fault localization model RATCHET-FL to localize the bug position by predicting the line number l of the buggy statement s_l in the buggy function c. With the buggy statement s_l identified by RATCHET-FL, the patch generation model RATCHET-PG of RATCHET generates the patch for it. RATCHET-PG is inspired by the recent works [30], [31], [34], we augment the buggy statement s_l by retrieving the most similar patch from a patch dataset D' to enhance the generation process.

A. RATCHET-FL*: Fault Localization Model*

Inspired by the redundancy assumption [26], [35] that the buggy code could be recurrent in programs, we propose to train the fault localization model RATCHET-FL with bugs collected from real-world projects, to predict the buggy statement of a given bug. The fault localization model RATCHET-FL takes as input a buggy function $c = \{s_1, s_2, ..., s_n\}$ to locate the single buggy statement s_l ($l \in [1, n]$) by predicting the line number l with the input c i.e., $l = f_{loc}(c)$, which is formulated as a multi-classification task. Fig. 2 overviews the architecture of RATCHET-FL, which consists of two sequential layers: *Embedding Layer* and *Feature Learning Layer*

1) **Embedding Layer:** An embedding layer *Embed* embeds code tokens into the unique numeric vector representations learned with the basic semantic information among tokens during the training phase. Code identifiers are named by developers with the combinations of natural language words/letters to represent the specific notion [36]. Concrete string literals and number literals are used to convey the specific values in code. The embedding of identifiers and literals will sharply increase the redundancy of vocabularies and noise the feature learning [37]–[39]. Thus, to address this challenge, we employ the camel-case and underscore naming convention to split code identifiers (e.g., the function name "getPacket" is split into two separated tokens: "get" and "packet"), and abstract string and number literals into placeholders "STRING" and "INT", respectively. It is commonly used in source code learning [40], [41] and learning-based APR [14], [16] to reduce the vocabulary size. For $\forall s_i \in c, s_i = \{t_1, t_2, ..., t_m\}$ (m is the number of tokens t in the i-th statement of the function c), the embedding layer can be represented with the following equation:

$$\mathbf{X} = \text{Embed}(t_1, t_2, ..., t_m) \tag{1}$$

where $\mathbf{x_k} \in \mathbf{X}$ is the k-th token representation and $\boldsymbol{X} \in \mathbb{R}^{m \times d}$, d is the dimension length.

2) **Feature Learning Layer:** $\forall \mathbf{x_k} \in \mathbf{X}$, we employ Bi-Directional LSTM (BiLSTM) [42] to learn the statement-level representation $\mathbf{r_{s_i}}$, which can be expressed as follows:

$$\begin{gathered}\mathbf{h_1}, \mathbf{h_2}, ..., \mathbf{h_m} = \text{BiLSTM}(\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_m}) \\ \mathbf{r_{s_i}} = [\mathbf{h}_{\mathbf{m}}^{\rightarrow}; \mathbf{h}_{\mathbf{1}}^{\leftarrow}]\end{gathered} \tag{2}$$

With the learnt statement-level representations $\mathbf{R} = \{\mathbf{r_{s_1}}, \mathbf{r_{s_1}}, ..., \mathbf{r_{s_n}}\}$ for the buggy function, we use softmax with two fully connected layers to give the probability of each statement that would be faulty. The cross-entropy loss function is selected for the learning process since we model this problem as a multi-class classification task.

B. RATCHET-PG*: Retrieval Augmented Transformer Model of Patch Generation*

In the community of deep learning-based tasks, the latest works [30], [31], [34], [43], [44] have proved that retrieval

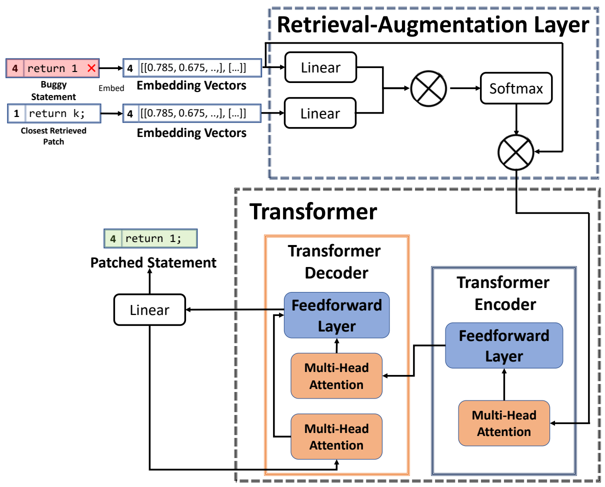


Fig. 3: Architecture of RATCHET-PG model.

techniques can yield competitive even better performance than the pure generation approaches (e.g., RNNs in code-to-text generation tasks like source code summarization [31] and commit message generation [30], [34]). We propose to explore the retrieval technique with deep learning for automated patch generation. In our preliminary study, we calculated the BLEU-4 score (a metric to measure the text similarity between the source and target input) between the closest retrieved patched statement and the exact patch of a buggy statement. The BLEU-4 score of retrieval augmented transformer is 0.6542 which is higher than the score (0.5547) of an LSTM-based Seq2Seq model. This study motivates us to design our retrieval-based patch generation model, RATCHET-PG, with two parts: ❶ retrieving the similar patch for the buggy statement and ❷ combining the retrieved patch into the transformer to generate the fixed patches (cf. CRP Retrieval and Patch Generation illustrated in Fig. 1).

1) Retrieving: For any single line buggy statement s_l in the buggy function c in the dataset D, RATCHET aims to retrieve the closest retrieved patch (CRP) from the dataset D', a specialized dataset composed of all possible patches. For simplification, we follow Liu et al's workaround [31] to use the patched statements in the training set as D' for RATCHET. The retrieving process can be formulated as below:

$$p' = \text{argmax}_{p' \in D'} \text{score}(s_l, p') \tag{3}$$

where p' represents the CRP retrieved from D', and $p' \neq s_l$. argmax is used to select the p' that is more similar to s_l than other patches in D'. In particular, we employ Lucene [45] to compute similarities between s_l and all patches in D'. The similarity score $score(s_l, p')$ can be computed as below:

$$\text{score}(s_l, p') = \text{coord}(s_l, p') + \text{qb}(s_l) + \text{sim}(s_l, p') + \text{db}(p') \tag{4}$$

where the function $\text{coord}(*)$ computes the score of overlapping query terms, $\text{qb}(*)$ and $\text{db}(*)$ refers to the query-boost factor and document-boost factor of Lucene, and $\text{sim}(*)$ represents the cosine similarity between s_l and p'. Boost factors can be specified for the concrete documents and query terms, thus the concrete documents or terms can have higher weights. We use default values (1) for query-boost and document-boost in our retrieval approach.

2) Retrieval-Augmented Transformer: The patch generation model RATCHET-PG takes as input the buggy statement s_l and generates the corresponding patch s_p by considering the closest retrieved patch p', the patch generation of RATCHET-PG can be expressed as $s_p = f_{gen}(s_l, p')$. Fig. 3 shows the overall architecture of RATCHET-PG, which consists of three parts: Retrieval-Augmentation Layer, Transformer Encoder and Transformer Decoder.

Retrieval-Augmentation Layer: Similar to RATCHET-FL, we embed the buggy statement s_l and its CRP p' into the vector representations using an embedding layer. Before that, we employ the same pre-processing method to split code identifiers. We then employ an attention layer to learn the attention vector $\boldsymbol{A}$ to compute the relevance between s_l and p'. For $s_l = \{t_1, t_2, ..., t_i\}, p' = \{\hat{t}_1, \hat{t_2}, ..., \hat{t}_j\}$ where i and j are the total number of sub-tokens in s_l and p', X and $\hat{X}$ are the embedded vectors of s_l and p' (cf. equation (1)), the attention matrix $\boldsymbol{A}$ can be expressed as below:

$$\begin{aligned} \boldsymbol{X} &= \text{Embed}(t_1, t_2, ..., t_i) \\ \hat{\boldsymbol{X}} &= \text{Embed}(\hat{t}_1, \hat{t}_2, ..., \hat{t}_j) \\ \boldsymbol{M} &= \text{ReLU}(\boldsymbol{W}^Q \boldsymbol{X}) \times \text{ReLU}(\boldsymbol{W}^R \hat{\boldsymbol{X}}^{\boldsymbol{T}}) \\ \boldsymbol{A} &= \text{softmax}(\boldsymbol{M}) \end{aligned} \tag{5}$$

where $\boldsymbol{W}^Q \in \mathbb{R}^{d \times d}$ and $\boldsymbol{W}^R \in \mathbb{R}^{d \times d}$ are learnable weights and ReLU is the rectified linear unit [46]. We multiply the attention matrix $\boldsymbol{A}$ with the retrieved patch features and add it to the original buggy features, which can be expressed as below:

$$\boldsymbol{Z} = \boldsymbol{X} + \boldsymbol{A}\hat{\boldsymbol{X}} \tag{6}$$

where $\boldsymbol{Z} \in \mathbb{R}^{i \times d}$ represents the final representations of s_l, i is the total token length of the buggy statement and d is the dimension length. Then we use a Transformer to proceed with the patch generation.

Transformer Encoder: The encoder takes as the input $\boldsymbol{Z}$ (the output of the Retrieval-Augmentation Layer) added with the positional encoding for learning. It is composed of a stack of N identical layers and each layer has two sub-layers. The first sub-layer is a multi-head attention layer and the second one is a fully connected feed-forward network. The residual connection followed by the layer normalization is used. The output of each sub-layer can be expressed as $\text{LayerNorm}(x + \text{Sublayer}(x))$.

The multi-head attention layer uses different head h to attend to information from representation subspaces at different positions. Each head employs scaled dot-product attention, which can be computed as below:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{Softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}})\boldsymbol{V} \tag{7}$$

where $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ represent the key, values, and queries, respectively. Multi-head attention representation can be computed by concatenating the results of scaled dot-product attention as below:

$$\begin{aligned}\text{MultiHead}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) &= \text{Concat}(\text{head}_\text{i},...,\text{head}_{\boldsymbol{h}})\boldsymbol{W}^O \\ \text{head}_i &= \text{Attention}(\boldsymbol{QW}_\text{i}^\text{Q},\boldsymbol{KW}_\text{i}^\text{K},\boldsymbol{VW}_\text{i}^\text{V})\end{aligned} \tag{8}$$

where $\boldsymbol{W}_i^Q \in \mathbb{R}^{d\times d_k}$, $\boldsymbol{W}_i^K \in \mathbb{R}^{d\times d_k}$, $\boldsymbol{W}_i^V \in \mathbb{R}^{d\times d_v}$ and $\boldsymbol{W}^O \in \mathbb{R}^{hd_v\times d}$ are model parameters and $d_k = d_v = d/h$. The feed-forward network sub-layer consists of multiple linear transformation and can be expressed as below:

$$\text{FFN}(\boldsymbol{x}) = \max(0, \boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2 \tag{9}$$

where $\boldsymbol{W}_1$, $\boldsymbol{W}_2$, $\boldsymbol{b}_1$, and $\boldsymbol{b}_2$ are weights and bias of the linear transformations.

Transformer Decoder: The decoder is composed of a stack of N identical layers. Different from the encoder layer, the decoder layer includes a third sub-layer that computes multi-head attention over the final output of the encoder layers. In the third multi-head attention sub-layer of the decoder, it uses the output of the encoder layers as $\boldsymbol{K}$ and $\boldsymbol{V}$, while $\boldsymbol{Q}$ is the output from the previous decoder layer. Similar to the transformer encoder, the decoder employs the same residual connection layer as the encoder. Finally, the last layer of the decoder outputs a matrix. We then compute the probability of tokens by using a linear layer with softmax to generate the probability of each token, which can be expressed as below:

$$\text{P(i)} = \text{softmax}(\boldsymbol{HW}) \tag{10}$$

where $\boldsymbol{H} \in \mathbb{R}^{l\times d}$, $\boldsymbol{W} \in \mathbb{R}^{d\times l_\text{vocab}}$, l is the input sequence length and l_vocab is the total vocabulary length.

C. Inference in Bug Fixing Pipeline

After the RATCHET-FL and RATCHET-PG are trained with the collected dataset, RATCHET can be used to infer patches in a normal bug fixing pipeline. For a given buggy function c, it is first fed into the fault localization model RATCHET-FL to compute the suspicious score for each statement and rank them with the scores. Specifically, for each statement s_i in c where $1 < i \leq n$ and n is number of statements, RATCHET-FL computes the suspicious score for each statement, and sorts statements with the scores to return the top-k suspicious statements, which is defined as $S^{sus} = \{s_1^{sus}, s_2^{sus}, ..., s_k^{sus}\}$.

For each suspicious statement $s_i^{sus} \in S^{sus}$, RATCHET employs Lucene to search the closest retrieval patch p_i' among all patches in the training set D and generates k patch candidates using the patch generation model RATCHET-PG, $s_p = f_{gen}(s_i^{sus}, p_i')$. Finally, we can get a set of patch candidates $S_p = \{s_{(p,1)}, s_{(p,2)}, ..., s_{(p,k)}\}$ for the buggy function c.

IV. EXPERIMENTAL SETUP

In this section, we first present the data preparation, list the selected baselines to compare with, and present the evaluation metrics, as well as provide the experimental settings.

TABLE I: Information of projects building RATCHET-DS.

Project	# Fix buggy Functions	Application Type
libxml2	285	Parser
tcpdump	155	Packet Analyzer
ImageMagick	23	Image Software
cURL	300	Networking/Data Transfer
Glibc	482	C Standard Library
openssl	662	Protocol
Asterisk	931	Communication Toolkit
PostgreSQL	1452	Database
Qemu	3422	Emulator
Wireshark	1196	Network/Packet Analyzer
FFmpeg	697	Multimedia Library
php-src	421	PHP Interpreter
Linux	33549	Operating System

A. Data Preparation

Existing benchmarks such as DeepFix dataset [21] and CodeFlaws [47] often have flaws in their design and quantity. The dataset of DeepFix consists of student programs with simple errors, such as replacing `";"` with `","`, and the number of tokens is far less than in real-world applications, which are unrealistic for in-the-wild programs. CodeFlaws has 3,902 defects, which is higher than other datasets (DBG-Bench [48], IntroClass [49], ManyBugs [49]). Nevertheless, the small amount of data is still a big challenge for deep learning approaches. Therefore, we curate a large benchmark RATCHET-DS from real-world projects that contain bug-patch pairs.

We selected 13 C/C++ open-source projects (shown in Table I) with two criteria: 1) Each project contains sufficient commits (from 5,058 to 951,181) that allow us to collect various data. 2) These projects range from a wide range of functionalities that provides our model data from different domains instead of focusing on a single area.

We crawled all the commits of the projects at October 2023, and employed a keyword filtering process [35], [50], [51] to extract bug-fixing commits by checking whether the message of a commit contains one of the bug-fixing related keywords (e.g., fix, solve, repair, bug, issue, problem, error, fault, vulnerable, CVE, exploit) After collecting the patches, we extract their functions and the bug-patch pairs, the patches with more than one-line changes are excluded from our dataset. Eventually, we curated the RATCHET-DS with 56,974 buggy functions with associated patches. In term of the multiple commits regarding one patch, we only collected programs that only required one commit, i.e., the buggy function before the fix commit and the patched function after the fix commit.

In our experiments, we randomly split the RATCHET-DS into three sets with 80%, 10%, and 10% to build a training set, a validation set, and a testing set.

To assess the effectiveness of RATCHET on fixing bugs and the benchmark-overfitting problem [52], RATCHET is also evaluated with the DrRepair Dataset [14] released in the literature. However, the DrRepair dataset cannot be directly used by RATCHET because of its collecting criteria and its design, we curate it by the following steps: 1) We randomly

TABLE II: Statistics of Dataset.

Dataset	Total	Training set	Validation set	Testing set
RATCHET-DS	56,974	45,575	5,695	5,704
DrRepair	57,344	45,875	5,734	5,735

selected two single-line bug-patch pairs for each program to satisfy our scenario; 2) We remove the duplicated pairs to avoid over-fitting and redundancy [53]. A summary of the RATCHET-DS and DrRepair dataset are shown in Table II.

B. Evaluation Baselines

We evaluate RATCHET in terms of the effectiveness of Fault Localization and Patch Generation. Thus, we compare the performance of RATCHET with two kinds of baselines against the state-of-the-art.

1) Baselines for Fault Localization: We select the following baselines for the comparison of fault localization.

Locus [54] employs the Vector Space Model (VSM) to locate bugs among their source code. Code Entity Model and Natural Language Model are used to embed the tokens and compute the similarity score between the bug reports and the code elements. We implemented a VSM model based on their approaches as Locus is not publicly available.

iFixR [25] employs TF-IDF with cosine similarity to locate the suspicious statements within the source file. Both Locus and iFixR employ information retrieval techniques in localizing faults in source code. Locus and iFixR are the two state-of-the-art retrieval-based fault localization techniques without considering the bug-triggering test cases. We thus select them as the two baselines for fault localization. In the replication of Locus and iFixR, we replace bug reports with commit messages as bug report is not available for all patches.

Transformer [55], based on the RATCHET-FL which is a classification task, we only employ the Transformer Encoder part to replace the BiLSTM in RATCHET-FL, assess the contribution of the Transformer, fairly comparing BiLSTM model from RATCHET-FL, on localizing fault positions.

Two state-of-the-art learning-based fault localization techniques (DeepFL [56] and DeepRL4FL [57]) are not considered as the baselines in this work since both of them leverage the test cases for the feature learning of fault localization.

2) Baselines for Patch Generation: We select the following baselines for the comparison of patch generation.

DeepFix [21] employs a Seq2Seq approach to generate patches for C/C++ programs. It utilizes LSTM networks to generate fixes based on each line of code in the program. We replicate it by implementing an LSTM-based seq2seq for generating single line patches with this baseline. We set our LSTM hidden dimension as 128 for this baseline.

SequenceR [15] uses Seq2Seq and Copy Mechanism to handle Out-of-Vocabulary words. It input the whole buggy function to generate a fixed line for the buggy function.

DrRepair [14] employs LSTM and GNN to capture the long-range dependencies that exist between the error location and the buggy statements. Then it generates the correct fix based on the localized buggy statements.

TABLE III: Results of Fault Localization with Acc@TopK

Methods	RATCHET-DS			DrRepair		
	Top1	Top3	Top5	Top1	Top3	Top5
Locus	6.89	21.48	45.04	5.61	23.96	45.67
iFixR	7.21	22.05	44.3	5.72	23.87	46.19
DrRepair	35.85	66.15	80.29	55	75.97	87.77
Transformer	34.62	62.52	76.14	79.74	89.36	94.14
RATCHET-FL	39.83	67.36	80.75	88.21	93.76	96.43

These three APR tools are state-of-the-art learning-based approaches, which are selected as the baselines in this study. DLFix [3] only works for Java programs. CoCoNut [16], and CuRe [17] are not replicable due to the limitation of their source code[2]. Thus, the three state-of-the-art learning-based APR tools are not presented in the comparison.

C. Evaluation Metrics

We employ Acc@TopK as the metric for the fault localization model, and BLEU-4 and RAcc for the patch generation model, respectively.

Acc@TopK Following existing works [54], [58] for the fault localization, we selected Acc@TopK as the metric. Specifically, given a set of localized suspicious statements $S^{sus} = \{s_1^{sus}, s_2^{sus}, ..., s_k^{sus}\}$, if the buggy statement s_l is within S^{sus}, it is located.

$$Acc@TopK = \frac{1}{n}\sum_{i=1}^{n}\delta(\mathrm{FRank}_i \leq k) \tag{11}$$

where n is the total samples in the dataset, δ is the function that returns 1 if the input is true, otherwise returns 0. FRank is the rank of the first hit result for the buggy statement s_l in the buggy function. In this work, we select k as 1, 3, 5. A higher Acc@TopK value indicates the better precision of fault localization.

BLEU-4 BLEU score [59] is a popular metric for NLP and source-code translation tasks [31]. It is used to evaluate the performance by computing text similarity between the output with the ground truth. We select BLEU-4 that employs four-gram in computing BLEU score to assess the similarity between the patches generated by APR tools and the ground truth. A higher BLEU-4 score indicates that the generated outputs are more similar to the ground truth.

Repair Accuracy (RAcc) As previously mentioned in Section III-A and Section III-B, our model outputs k plausible patches S_p. To assess the performance of generating patches of RATCHET, we further check whether the patch within S_p is identical to the ground-truth patch, and assess to what extent accurate patches can be generated by RATCHET. We define the RAcc as below:

$$RAcc = \frac{1}{n}\sum_{i=1}^{n}\delta(\mathrm{Hit_i}) \tag{12}$$

where δ is the function that returns 1 if the input is true, otherwise, returns 0. Hit is the function to express whether $y \in S_p$, where y is the ground-truth.

[2]https://github.com/lin-tan/CURE/issues/4

D. Experimental Settings

In the fault localization model, we employ a 2-layer BiLSTM and set word embedding size and LSTM hidden size as 128. The dropout [60] is set to 0.3 and the learning rate as 0.001. The batch size is 16 with Adam [60] optimizer. For the patch generation model, we employ Transformer-based model with a dimension size of 128 and set its feed-forward layer dimension as 128. We use a dropout of 0.2 for its positional encoding and transformer layer dropout for better performance. We use a 4-layer transformer encoder and decoder to encode and generate the output sequences. Furthermore, we use 4 transformer heads for our multi-head attention. Similarly, Adam [61] is used as our optimizer. We trained our models using the training set, and tune them with the validation set. The testing set is used to evaluate the performance of our model. For both RATCHET-FL and RATCHET-PG, we use patience of 10 epochs and conduct our experiment on 3 Nvidia-Tesla-V100 graphic cards. The experiments on RATCHET-PG and RATCHET-FL took an average of 7 hours and 11 minutes in training respectively.

V. EVALUATION AND RESULTS

In this work, we evaluate our approach by comparing against the baselines with the two datasets, RATCHET-DS, and DrRepair Dataset. We aim to answer the following research questions:

- **RQ1:** To what extent real-world bugs can be accurately localized by RATCHET-FL?
- **RQ2:** Can RATCHET-PG generate correct patches for the real-world bugs localizaed by RATCHET-FL?
- **RQ3:** Would the patch generation ability of RATCHET-PG be affected by different retrieved contexts?
- **RQ4:** Is RATCHET sensitive to the fault localization setting for fixing bugs?

A. *RQ1: Performance of* RATCHET-FL *on Fault Localization*

We evaluate the performance of RATCHET-FL with the two datasets (RATCHET-DS and DrRepair dataset) and compare its performance against four different baselines. The corresponding results are shown in Table III. For a clear understanding of the fault localization performance, Fig. 4 presents the distribution on code lines of buggy functions in the two datasets. SBFL techniques are not considered in the comparative experiment due to the lack of bug-triggering test cases in both datasets. Comparing with the two information retrieval approaches (Locus and iFixR), RATCHET-FL outperforms them with more 32.6-45.9 percentage points in the RATCHET-DS and with more 50.2-82.6 percentage points in DrRepair dataset at the Acc@Top1, Acc@Top3, and Acc@Top5 metrics. The low performance of Locus and iFixR might be caused by the different retrieval indexes. The original Locus and iFixR employ bug reports for localization. However, it is infeasible to identify bug reports for a massive dataset. Therefore, we substitute the bug reports with commit messages. Actually, many bugs are not associated with any bug reports [56]. Nevertheless, unlike other fault localization tools [25], [56], [57], RATCHET-FL does not require any bug reports or bug-triggering test cases, which allows RATCHET-FL to locate faults at the early software development stage.

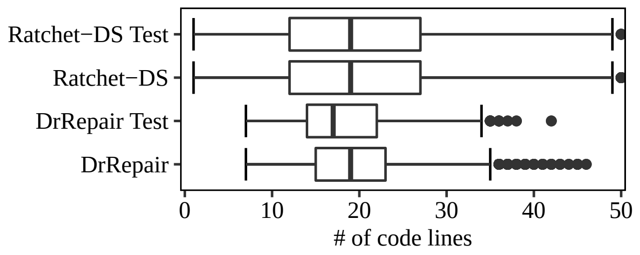


Fig. 4: Distribution on the code lines of buggy functions.

Comparing with two learning-based baselines of fault localization, RATCHET-FL achieves higher accuracy at the three metrics after evaluating them on RATCHET-DS and DrRepair Dataset. It implies that the simple bi-directional LSTM (BiLSTM) model of RATCHET-FL outperforms the DrRepair and Transformer Encoder for fault localization. BiLSTM model of RATCHET-FL can effectively capture the feature from source code for fault localization, although DrRepair leverages the combination of LSTM and GNN to expose bug position with the source code and compiling message. Furthermore, it is not so feasible to expect compiler messages of real-world bugs that could take hours to compile the whole big program for a single bug, especially in deep-learning approaches.

Notably, RATCHET-FL performs better fault localization in the DrRepair dataset than the RATCHET-DS, with an increase of 48.4% in Accuracy@Top1, 26.4% in Accuracy@Top3 and 15.7% in Accuracy@Top5. DrRepair dataset consists of the bugs in student programs and their mutation, which are much simpler than the real-world bugs in the RATCHET-DS. RATCHET-DS is curated with 56,974 functions from real-world programs, which shows a higher diversity than bugs from student programs. Localizing positions for in-the-lab bugs poses fewer challenges than in-the-wild ones.

✍ ►**RQ1**◄ The simple BiLSTM model-based RATCHET-FL achieves promising performance on localizing positions of in-the-lab and in-the-wild bugs without considering the bug reports and bug-triggering test cases, which outperforms state-of-the-art retrieval based and learning-based approaches. Note that the performance of fault localization for in-the-lab bugs might not be so practical for real-world bugs.

B. *RQ2: Performance of* RATCHET *on Program Repair*

We assess the program repair performance of RATCHET and compare it with three deep learning baselines. DeepFix [21] and SequenceR [15] requires perfect fault localization setting. Here the perfect means that the location of the buggy statement is known. To avoid the bias from the different fault localization settings, we employ RATCHET-FL to localize the bug positions for DeepFix and SequenceR before generating the patches. DrRepair encompasses both fault localization and

TABLE IV: Results of Automated Repair.

Methods	RATCHET-DS BLEU-4	RATCHET-DS RAcc	DrRepair Dataset BLEU-4	DrRepair Dataset RAcc
RATCHET-FL +DeepFix	15.51	7.55	24.18	37.03
RATCHET-FL +SequenceR	6.42	4.92	7.64	18.30
DrRepair	0.04	0.02	11.01	21.19
No FL + RATCHET-PG	19.38	2.51	19.10	1.29
Perfect FL + RATCHET-PG	65.73	19.71	85.73	46.56
RATCHET	18.43	19.51	25.52	46.38

patch generation into one deep learning model, thus its fault localization is unchanged in this experiment.

Table IV shows the experimental results (the higher metric values, the better performance, cf. Section IV-C) of repairing bugs. DeepFix and DrRepair achieve overwhelming performance on the DrRepair dataset than the RATCHET-DS. The two learning-based APR tools were inspired by the students' programs, were proposed and evaluated on the in-the-lab dataset (e.g., bugs from students' programs in DrRepair dataset). It infers that building the learning-based ARP models from the in-the-lab dataset could be impractical for solving real-world bugs. RATCHET outperforms all of the three learning-based APR baselines at generating patches for bugs in both RATCHET-DS and DrRepair dataset. RATCHET also achieves better results on generating patches for in-the-lab bugs than in-the-wild ones, which is consistent with the performance of fault localization (cf. Section V-A) since the bugs in students' programs from DrRepair dataset are much simpler than real-world bugs. We infer that the program repair performance of RATCHET might be benefited from our proposed retrieval-based patch generation model that allows the transformer to generate better patches by learning on correct code sequences. And the retrieval contexts allow the generation attends to tokens that are not within their inputs.

✍ ►**RQ2**◄ RATCHET is effective in generating patches for in-the-lab and in-the-wild bugs, which outperforms the three state-of-the-art learning-based APR baselines.

C. *RQ3: Impact of Retrieval Contexts on* RATCHET-PG

Our proposed retrieval-augmented transformer model learns from correct patches, which contributes RATCHET to fix more bugs than the state-of-the-art learning-based APR baselines as observed from RQ2. As the default configuration, we selected CRP as the retrieval context. In this section, we investigate the impact of different retrieval contexts on the generation of patches. Specifically, we selected four different retrieval contexts and evaluate their performances: Closest Retrieval Buggy Line (CRBL), Closest Retrieval Buggy Function (CRBF), Closest Retrieval Patch Function (CRPF), and one experiment without any context (w/o C).

The experimental results are presented in Table V. Comparing on the BLEU-4, RATCHET-PG with the CRP context outperforms other context settings on both RATCHET-DS

TABLE V: Results of patch generation with various contexts.

Context	RATCHET-DS BLEU-4	RATCHET-DS RAcc (Num)	DrRepair Dataset BLEU-4	DrRepair Dataset RAcc (Num)
w/o C	0.1815	0.1832 (1045)	0.2533	0.4397 (2522)
CRBL	**0.1846**	0.1921 (1096)	0.2509	0.4333 (2485)
CRBF	0.1737	0.1649 (941)	0.2480	0.4186 (2401)
CRPF	0.1821	0.1823 (1040)	0.2501	0.4292 (2462)
CRP	0.1843	**0.1949 (1112)**	**0.2552**	**0.4598 (2637)**

*"(Num)" denotes the number of bugs fixed by RATCHET-PG with the corresponding context setting.

(except CRBL[3]) and DrRepair Dataset. Buggy statements and functions are incorrect, which results in the RATCHET-PG model learns from negative examples for patch generation. We observe that using closest buggy or patch functions as the contexts can negatively impact the patch generation of RATCHET-PG since the two kinds of contexts contain more tokens than CRP. On average, there are 13 tokens on the line-level contexts, whereas there are 122 tokens on the function-level contexts. The sharply increased tokens cause a long-range dependency problem and could result in the model focusing on the wrong information. The CRBF context shows the worst performance among all contexts. We infer that the longer sequence of function-level contexts leads to the decrements of CRPF context against the CRP context. Intuitively, learning from closest retrieval patches boosts the generation model to generate syntactically correct patches. Our qualitative analysis in Section VI-A further shows that the RATCHET-PG model can alleviate the missing token problem by finding and attending to tokens that do not exist in the input source code.

✍ ►**RQ3**◄ The context of closest retrieval patches can contribute to generating correct patches for bugs. Such contribution is not so positive for the context from the closest retrieval buggy/patch functions, since long sequences of function-level contexts are not as effective as the statement-level contexts for learning correct patches to generate useful and effective patches.

D. *RQ4: Significance of Fault Localization.*

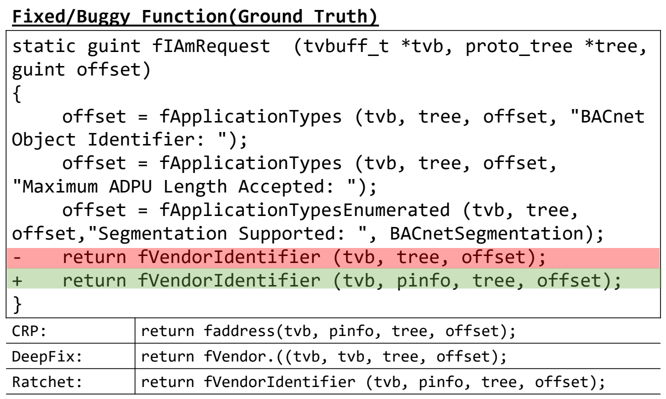
As reported by Liu et al. [23], fault localization setting has a direct impact on the performance of patching for APR tools. In this section, we investigate the impact of fault localization on repairing bugs for RATCHET. Specifically, we train a model with buggy functions instead of buggy statements and aim to generate patches based on buggy functions (i.e., without fault localization (No FL) in Table VI). We proceed a perfect fault localization setting for RATCHET-PG the same as CoCoNut [16], CuRE [17], and SequenceR [15].

Table VI presents the bug-fixing results of RATCHET-PG with different fault localization settings. With the perfect fault localization setting, RATCHET-PG achieves the best performance on generating patches, which outperforms RATCHET-

[3]This implies that the learning model generates patches with higher similarity, but cannot ensure the correctness of the generated patches.

TABLE VI: Impact of Fault Localization on Program Repair.

Fault Localization (FL) Settings	RATCHET-DS BLEU-4	RATCHET-DS RAcc	DrRepair Dataset BLEU-4	DrRepair Dataset RAcc
No FL + RATCHET-PG	19.38	2.51	19.10	1.29
Perfect FL + RATCHET-PG	65.73	19.71	85.73	**46.56**
RATCHET	18.57	19.21	25.52	46.38



Fixed/Buggy Function(Ground Truth)

```
static guint fIAmRequest  (tvbuff_t *tvb, proto_tree *tree,
guint offset)
{
    offset = fApplicationTypes (tvb, tree, offset, "BACnet
Object Identifier: ");
    offset = fApplicationTypes (tvb, tree, offset,
"Maximum ADPU Length Accepted: ");
    offset = fApplicationTypesEnumerated (tvb, tree,
offset,"Segmentation Supported: ", BACnetSegmentation);
-    return fVendorIdentifier (tvb, tree, offset);
+    return fVendorIdentifier (tvb, pinfo, tree, offset);
}
```

CRP:	`return faddress(tvb, pinfo, tree, offset);`
DeepFix:	`return fVendor.((tvb, tvb, tree, offset);`
Ratchet:	`return fVendorIdentifier (tvb, pinfo, tree, offset);`

Fig. 5: Ground-truth and generated patches for bug Wireshark:809fb76.

PG with "No Localization" and normal fault localization settings. It is consistent with APR tools studied in the literature [19]. On the other hand, although the BLEU-4 values of RATCHET are not as good as the perfect fault localization setting for RATCHET-PG, its RAcc values on the two datasets are just a bit lower than the perfect fault localization setting. The results indicate that RATCHET is not so sensitive to the fault localization setting for generating correct patches.

✍ ►**RQ4**◄ Different fault localization settings can impact the performance of RATCHET-PG in terms of returning better BLEU-4 value, but the impact is not so great on generating correct patches for RATCHET.

VI. Discussion

A. *Qualitative Case Studies*

We present one cases of patches generated by RATCHET with high quality in Fig. 5, where the "Buggy Function" shows the buggy function (it is the input of related models) and the related ground-truth patch, and "CRP" shows the closest retrieved patch identified by Lucene. "DeepFix" and "RATCHET " denote the patches generated by themselves, respectively.

Fig. 5 shows a patch excerpted from commit 809fb76 in Wireshark. In the ground truth patch, an additional argument "`pinfo`" is inserted into the invocation of function "`fVendorIdentifier`". DeepFix predicts that this bug should be fixed by adding an argument to the function, but it could not accurately determine the correct function name and arguments. Indeed, the generated patch is syntactically similar (with a BLEU-4 value) to the ground-truth patch but cannot fix the bug. On the contrary, RATCHET can determine the function call and structure of the return statement by our proposed retrieval augmented model. With the closest retrieval patch (CRP) that provides the correct parameters for the patch, RATCHET finally generates a correct patch for the bug that is identical to the ground truth.

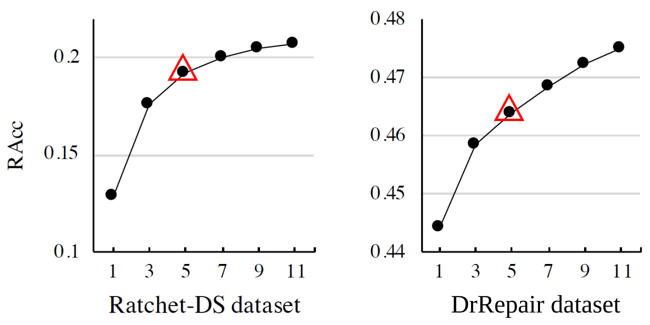


Fig. 6: Impact of K's values on RATCHET.

B. *Selection of K's Value for* RATCHET

As mentioned in Section III-C, RATCHET generates k patch candidates by using the top-k suspicious statements. The number of patch candidates directly impacts the effectiveness of fixing bugs [19]. And, the model performance might be affected if the value of k is not large enough. Therefore, we investigate the impact of different values of k on RATCHET. Specifically, we apply different values ({1, 3, 5, 7, 9, 11}) of k for RATCHET to assess its performance.

Fig. 6 presents the RAcc results of RATCHET setting with different k values for two datasets. As the value of k increases, the repair accuracy of RATCHET on RATCHET-DS increases sharply before reaching a plateau of approximately 0.2 when k's value increases to 5. As the value of k increases from 5 to 7, the repair accuracy increases marginally, with an increment of less than 1%. RATCHET presents a similar trend on DrRepair dataset. Intuitively, more correct patches can be captured within a large number of patch candidates. Therefore, the value of k should be as high as possible. However, the trade-off is the increased amount of plausible patches with the higher k value, which hinders the validation of correct patches [52] and impacts the efficiency of fixing bugs [19]. Therefore, the number of generated patch candidates should be as small as possible without a sharp decrease in RAcc. As illustrated in Fig. 4, most buggy functions have ∼20 statements. RATCHET-FL presents auspicious performance on exposing bugs with the top-5 most suspicious statements, hence, $k = 5$ is also a reasonable setting for fault localization. To sum up, RATCHET is set with $k = 5$ in this study.

C. *THREATS TO VALIDITY*

Several internal threats lie in this work. First, we focus on single-line bugs; although statistics from [62] state that almost 33% of bugs represent a single statement, there is large room to extend future works to more complex and multiple position program repair. Another threat arises from retrieving historical repair commits based on the context of buggy at the semantic level. Irrelevant repair commits might be retrieved, which can interfere with model training and lead to the generation of incorrect patches. Furthermore, the performance of neural models can vary significantly with different hyperparameter settings. Finding the optimal settings is often expensive and time-consuming. Although we conduct a grid search to identify settings offering relatively better performance, we cannot guarantee that the current settings are the best.

One of the external threats lies in the evaluation of our approach. Our approach has been evaluated solely on a vulnerability dataset in the C language. We have not explored its generalization to vulnerability datasets in other programming languages or CWEs like libraries. The current approach is language-specific, employing features in the template-mining module, which limits its direct application to other programming languages. Future work will focus on adapting the approach to be language-agnostic.

VII. Related Works

A. Fault Localization

In many automated program repair works [3], [4], [7], [8], [11], [12], SBFL techniques are used to localize bugs and errors in source codes. Ochiai [9] and Tarantula [10] are two popular algorithms in computing suspicious scores for each statement in the programs. GZoltar [24] is used for automatic testing and fault localization in Java projects. However, they rely on test cases for the computation of the suspicious score. Some works employ information retrieval techniques to locate faults. Their granularity ranges from buggy source files [54] to buggy statements [25]. Locus [54] constructs two models using code entity tokens and natural language tokens. By giving different weights to the models, it learns the cosine similarity between bug reports and statements. iFixR employs a TF-IDF model to compute the similarity score between reports and source code. Cosine similarity scores are used as the suspicious score for ranking the suspicious statements. Recent works have started to employ deep learning approaches in fault localization, such as DrRepair [14], which uses bidirectional LSTM and compiler messages to localize errors and generate fixed statements for the error. Nevertheless, bug reports and compiler messages are difficult to collect on a large scale in real-world open-sourced projects. In comparison, by our localization experiments, we have confirmed that the simple bidirectional LSTM could have better performance than the Transformer and we directly utilize LSTMs for localization.

B. Automated Program Repair

Automated program repair works can be grouped into three categories: Search-based, Semantic-based, and Deep Learning-based approaches. Earlier works involve a search-based approach to finding patches to fix programs. GenProg [7], [8] employs genetic programming and mutation-based operations to search for patches that fix C Programs. SemFix [11] fixes the program by making changes to a single statement and inferring constraints through the execution path. Some works augmented the search space through prioritization and context. Prophet [6] prioritizes patches through learning on candidates patches, while CapGen [4] learns from the history of the source code. Recent analysis [52], [63] has shown that generate-and-validate approaches suffer from test cases overfitting and plausible patch overloading. SemFix [11] makes changes to a single statement and infers constraints through the execution path. FixMiner [64] learns from historical fix templates to generate patches for existing bugs. More works employ a deep learning approach in learning patches for buggy programs. This can be attributed to the growing availability of open-source codebases, practical programming languages, and learning models. DLFix [3] employs graph-based learning on Abstract Syntax Tree to generate fixes for Java Program, while Deepfix [21], SequenceR [15], CoCoNut [16], and CuRe [17] employ a sequence-to-sequence techniques in generating patches. SynFix [65] repairs syntax errors through learning on language models and compiler errors. One of the limitations of these approaches is that they do not incorporate fault localization in their model. The performance of the fault localization greatly affects the performance of the generation model. Although DrRepair [14] includes fault localization, it relies on compiler messages for more contexts. Furthermore, in our work, we propose an automated program repair model based on the Transformer architecture and further incorporate the retrieved code snippets to improve the accuracy of the generated statement.

VIII. Conclusion

In this paper, we present our dual-model approach, RATCHET, that consists of fault localization RATCHET-FL and patch generation RATCHET-PG. RATCHET-FL localizes fault in buggy function by capturing the semantic of the buggy statements. Furthermore, RATCHET-FL does not require additional software artifacts in localizing the fault in the source code. RATCHET-PG is applied with our proposed retrieval-augmented transformer, which allows generating better patches for fixing bugs. The extensive experimental results confirm that RATCHET can outperform baselines in both fault localization and patch generation, and RATCHET is not so sensitive to the fault localization setting. Future study consists of how to incorporate more external information such as the compiler diagnostic feedback to further improve the system performance.

IX. Acknowledgments

This work is partially supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), and the Singapore Ministry of Education Academic Research Fund Tier 1 (RG12/23). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] Github, "The 2020 state of the octoverse," Github, Report P-19, 2020.

[2] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 492–501. [Online]. Available: https://doi.org/10.1145/1134285.1134355

[3] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 602–614.

[4] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–11. [Online]. Available: https://doi.org/10.1145/3180155.3180233

[5] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring program transformations from singular examples via big code," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 255–266.

[6] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 298–312. [Online]. Available: https://doi.org/10.1145/2837614.2837617

[7] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 3–13.

[8] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[9] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[10] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 467–477. [Online]. Available: https://doi.org.remotexs.ntu.edu.sg/10.1145/581339.581397

[11] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 772–781.

[12] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 416–426. [Online]. Available: https://doi.org/10.1109/ICSE.2017.45

[13] S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury, "Test-equivalence analysis for automatic patch generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 4, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3241980

[14] M. Yasunaga and P. Liang, "Graph-based, self-supervised program repair from diagnostic feedback," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 119. PMLR, 2020, pp. 10 799–10 808. [Online]. Available: http://proceedings.mlr.press/v119/yasunaga20a.html

[15] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[16] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 101–114. [Online]. Available: https://doi.org/10.1145/3395363.3397369

[17] N. Jiang, T. Lutellier, and L. Tan, "CURE: code-aware neural machine translation for automatic program repair," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 1161–1173. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00107

[18] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 702–713.

[19] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 615–627.

[20] M. Tufano, C. Watson, G. Bavota, M. di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 832–837.

[21] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, ser. AAAI'17. AAAI Press, 2017, p. 1345–1351.

[22] X. Li, S. Liu, R. Feng, G. Meng, X. Xie, K. Chen, and Y. Liu, "Transrepair: Context-aware program repair for compilation errors," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[23] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 102–113.

[24] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 378–381.

[25] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, "Ifixr: Bug report driven program repair," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 314–325. [Online]. Available: https://doi.org/10.1145/3338906.3338935

[26] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 306–317. [Online]. Available: https://doi.org/10.1145/2635868.2635898

[27] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*. ACM, 2014, pp. 492–495. [Online]. Available: https://doi.org/10.1145/2591062.2591114

[28] M. Allamanis, "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*, 2019, pp. 143–153.

[29] S. Liu, Y. Li, X. Xie, W. Ma, G. Meng, and Y. Liu, "Automated commit intelligence by pre-training," *ACM Transactions on Software Engineering and Methodology*.

[30] S. Liu, C. Gao, S. Chen, N. L. Yiu, and Y. Liu, "Atom: Commit message generation based on abstract syntax tree and hybrid ranking," *IEEE Transactions on Software Engineering*, 2020.

[31] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid GNN," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=zv-typ1gPxA

[32] S. Benton, X. Li, Y. Lou, and L. Zhang, "On the effectiveness of unified debugging: An extensive study on 16 program repair systems," in *35th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2020, pp. 907–918. [Online]. Available: https://doi.org/10.1145/3324884.3416566

[33] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, "Can automated program repair refine fault localization? a unified debugging approach," in *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual*

Event, USA, July 18-22, 2020. ACM, 2020, pp. 75–87. [Online]. Available: https://doi.org/10.1145/3395363.3397351

[34] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: how far are we?" in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 373–384.

[35] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. L. Traon, "A closer look at real-world patches," in *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution*. IEEE Computer Society, 2018, pp. 275–286. [Online]. Available: https://doi.org/10.1109/ICSME.2018.00037

[36] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon, "Learning to spot and refactor inconsistent method names," in *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering*. IEEE, 2019, pp. 1–12.

[37] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 38–49. [Online]. Available: https://doi.org/10.1145/2786805.2786849

[38] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code != big vocabulary: Open-vocabulary models for source code," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1073–1085. [Online]. Available: https://doi.org/10.1145/3377811.3380342

[39] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[40] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/49265d2447bc3bbfe9e76306ce40a31f-Paper.pdf

[41] S. Liu, W. Ma, J. Wang, X. Xie, R. Feng, and Y. Liu, "Enhancing code vulnerability detection via vulnerability-preserving data augmentation," in *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2024, pp. 166–177.

[42] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[43] F. F. Xu, Z. Jiang, P. Yin, and G. Neubig, "Incorporating external knowledge through pre-training for natural language to code generation," in *Annual Conference of the Association for Computational Linguistics*, 2020.

[44] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[45] A. Lucene. Apache lucene - welcome to apache lucene. [Online]. Available: https://lucene.apache.org/

[46] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

[47] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 180–182.

[48] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 117–128. [Online]. Available: https://doi.org/10.1145/3106237.3106255

[49] C. L. Goues, N. J. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of C programs," *IEEE Trans. Software Eng.*, vol. 41, no. 12, pp. 1236–1256, 2015. [Online]. Available: https://doi.org/10.1109/TSE.2015.2454513

[50] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings of the International Conference on Software Maintenance*. San Jose, California, USA: IEEE, 2000, pp. 120–130.

[51] K. Pan, S. Kim, and E. J. W. Jr., "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[52] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, "A critical review on the evaluation of automated program repair systems," *Journal of Systems and Software*, vol. 171, p. 110817, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302156

[53] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[54] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 262–273. [Online]. Available: https://doi.org/10.1145/2970276.2970359

[55] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[56] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 169–180.

[57] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization with code coverage representation learning," in *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, 2021, pp. 661–673.

[58] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 14–24.

[59] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040

[60] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[61] D. P. Kingma and J. Ba, "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*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[62] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? the manysstubs4j dataset," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 573–577.

[63] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 532–543. [Online]. Available: https://doi.org/10.1145/2786805.2786825

[64] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. L. Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *CoRR*, vol. abs/1810.01791, 2018. [Online]. Available: http://arxiv.org/abs/1810.01791

[65] T. Ahmed, N. R. Ledesma, and P. T. Devanbu, "SYNFIX: automatically fixing syntax errors using compiler diagnostics," *CoRR*, vol. abs/2104.14671, 2021. [Online]. Available: https://arxiv.org/abs/2104.14671