# Is this Change the Answer to that Problem?

## Correlating Descriptions of Bug and Code Changes for Evaluating Patch Correctness

Haoye Tian
haoye.tian@uni.lu
University of Luxembourg
Luxembourg

Xunzhu Tang
xunzhu.tang@uni.lu
University of Luxembourg
Luxembourg

Andrew Habib
andrew.a.habib@gmail.com
University of Luxembourg
Luxembourg

Shangwen Wang
wangshangwen13@nudt.edu.cn
National University of Defense Technology
China

Kui Liu*
brucekuiliu@gmail.com
Huawei
China

Xin Xia
xin.xia@acm.org
Huawei
China

Jacques Klein
jacques.klein@uni.lu
University of Luxembourg
Luxembourg

Tegawendé F. Bissyandé
tegawende.bissyande@uni.lu
University of Luxembourg
Luxembourg

## ABSTRACT

Patch correctness has been the focus of automated program repair (APR) in recent years due to the propensity of APR tools to generate overfitting patches. Given a generated patch, the oracle (e.g., test suites) is generally weak in establishing correctness. Therefore, the literature has proposed various approaches of leveraging machine learning with engineered and deep learned features, or exploring dynamic execution information, to further explore the correctness of APR-generated patches. In this work, we propose a novel perspective to the problem of patch correctness assessment: *a correct patch implements changes that "answer" to a problem posed by buggy behavior.* Concretely, **we turn the patch correctness assessment into a Question Answering problem**. To tackle this problem, our intuition is that natural language processing can provide the necessary representations and models for assessing the semantic correlation between a bug (question) and a patch (answer). Specifically, we consider as inputs the bug reports as well as the natural language description of the generated patches. Our approach, Quatrain, first considers state-of-the-art commit message generation models to produce the relevant inputs associated to each generated patch. Then we leverage a neural network architecture to learn the semantic correlation between bug reports and commit messages. Experiments on a large dataset of 9 135 patches generated for three bug datasets (Defects4j, Bugs.jar and Bears) show that Quatrain achieves an AUC of 0.886 on predicting patch correctness, and recalling 93% correct patches while filtering out 62% incorrect patches. Our experimental results further demonstrate the influence of inputs quality on prediction performance. We further perform experiments to highlight that the model indeed learns the relationship between bug reports and code change descriptions for the prediction. Finally, we compare against prior work and discuss the benefits of our approach.

## CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

## KEYWORDS

Patch Correctness, Program Repair, Question Answering, Machine Learning

**ACM Reference Format:**
Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F. Bissyandé. 2022. Is this Change the Answer to that Problem? : Correlating Descriptions of Bug and Code Changes for Evaluating Patch Correctness. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556914

## 1 INTRODUCTION

Generate-and-validate techniques have achieved success in automatic program repair (APR) by yielding valid patches for a large number of defects in several benchmarks [13, 30, 38, 58, 61, 63]. While such techniques are commonplace, their adoption by the industry faces a critical concern with respect to their practicality: state-of-the-art approaches tend to generate patches that overfit the weak oracles (e.g., test suites) [26, 31, 51, 64]. Indeed, in practice,

*Corresponding author.

This work is licensed under a Creative Commons Attribution International 4.0 License.

*ASE '22, October 10–14, 2022, Rochester, MI, USA*
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3556914

```
Missing type-checks for var_args notation
```

**(a) Title of the bug report.**

```
check var_args properly
```

**(b) The commit message of the developer's patch.**

**Figure 1: The bug report of Closure-96 from Defects4J and the corresponding commit message of the developer's patch.**

patches validated by test cases are only plausible. Most of them will be manually found by practitioners to be incorrect [8, 29, 30].

Research on automatic assessment of patch correctness has been prolific in recent years [48, 49, 52, 55]. We identify mainly two categories leveraging either static or dynamic information. In the first category, only static information is leveraged to decide on patch correctness. For example, Ye *et al.* [60] have manually crafted static features of code changes that can be used for training a machine learning (ML) based classifier of patch correctness. Similar approaches based on deep representation learning have been proposed [48]. More recently, Tian *et al.* [47] proposed a system where correctness is decided by checking the static similarity of failing test cases vs the similarity of code changes. In the second category, traces of dynamic execution of test suites are leveraged for correctness evaluation. To predict patch correctness, Xiong *et al.* [54] check the behavioral change of failing test case executions. Shariffdeen *et al.* [44] relied on concolic execution to traverse test inputs and patch spaces to reduce the number of patch candidates.

Despite the promising results achieved by the aforementioned approaches to patch correctness assessment, we identify one fundamental issue and one opportunity that open roads to the new research direction studied in this work. As a fundamental issue, we note that state-of-the-art approaches generally assess patch correctness by reasoning mostly about the code changes, and sometimes also about the test case. However, **the bug itself, which is targeted by the generated patch, is seldom explicitly investigated**. Yet, patches are written to address a specific buggy behavior. As an opportunity, we note that bug reports, while informal, may offer an explicit description of the bug, which can be leveraged to assess patch correctness.

To the best of our knowledge, no prior work has investigated the problem of patch correctness as a question-answering problem. We follow the intuition that when a code base maintainer is presented with a patch, the suggested changes are evaluated with respect to the reported bug. That bug is therefore a question. Bug reports, with their natural language description (cf. Example of Figure 1a), typically pose the problem. The code changes implementing a patch offer an answer to the problem. The commit message describing these changes (cf. Example of Figure 1b) typically presents the solutions to the maintainer. The maintainer can then immediately perceive whether the solution (patch) would be relevant to the problem (bug). This scenario of patch validation by human maintainers may appear naive since there are other aspects that developers consider, including whether the bug is real, whether the changes are riskier, etc. Nevertheless, this constitutes a first screening process that we aim to automate by leveraging recent advances in natural language processing (NLP) and machine learning (ML).

**This paper.** We explore the feasibility of leveraging a deep NLP model to assess the semantic correlation between a bug description and a patch description, towards predicting patch correctness for automated program repair. Our main contributions are as follows:

❶ We perform a preliminary validation study to demonstrate that bug and patch descriptions are correlated within a dataset of developer submitted patches. This hypothesis validation constitutes a first finding that opens a novel direction for patch correctness studies using bug artifacts.

❷ We formulate the patch correctness assessment problem as a question answering problem and propose Quatrain (Question Answering for Patch Correctness Evaluation), a supervised learning approach that exploits a deep NLP model to classify the relatedness of a bug report with a patch description.

❸ We extensively evaluate the effectiveness of Quatrain to identify correct patches as well as filter out incorrect patches among a dataset of 9,135 plausible patches (written by developers or generated by APR tools). Our evaluation further compares Quatrain to state-of-the-art dynamic [54] and static [48] approaches, and demonstrates that Quatrain achieves comparable or better performance in terms of AUC, F1, +Recall and -Recall.

❹ We conduct an analysis of the impact of inputs quality on the prediction performance. In particular, we show that the software engineering committee could benefit from extended research into the direction of patch summarization (a.k.a. commit message generation).

**Availability.** Our artifact, code, and dataset are publicly available at: https://github.com/Trustworthy-Software/Quatrain.

The remainder of this paper is organized as follows. Section 2 provides information on the related work, presents our intuition and summarizes validation data on the hypothesis of our work. Section 3 overviews our proposed approach. Experimental setup and results are then described in Sections 4 and 5 respectively. We provide discussions in Section 6 and conclude in Section 7.

## 2 RELATED WORK & HYPOTHESIS

In this section, we describe the related work to highlight the relevance of our work and the novelty of our approach. Then we validate the hypothesis that Quatrain builds on.

### 2.1 Related work

***Patch correctness.*** Test suites are widely used as the oracle to validate the correctness of generated patches in APR. Nonetheless, given that a test suite is an imperfect (i.e., non-comprehensive) specification of the program correctness, a patch that passes the test suite may still be incorrect: such patch is referred to in the literature as an *overfitting* patch [31, 41]. With the increasing interest in program repair, a number of research efforts have been undertaken towards identifying such overfitting patches, aiming at mitigating this limitation in APR [52]. In general, existing approaches can be split into two categories based on whether they need to execute test cases: static approaches rely on simple heuristics that are summarized from expert knowledge or extracted with learning-based techniques that capture deep features of the patch. For instance, Tan *et al.* [46] enumerated several code change rules that help identify overfitting patches as those that violate them. Ye *et al.* [60] trained a machine learning based classifier, ODS, which is based on

manually-crafted features specially designed for assessing patch correctness (e.g., presence of binary arithmetic operators). Tian *et al.* [48] investigated the feasibility of utilizing representation learning techniques (e.g., CC2Vec [15] and code2vec [2]) for comparing overfitting and correct patches. In contrast, the dynamic approaches utilize extra information obtained during test execution, such as the execution results on newly-generated test cases [53, 59, 62], the test runtime trace [54], and the program invariant inferred from the execution [56]. Specifically, Ye *et al.* [62] used two test generation tools (i.e., Evosuite [11] and Randoop [39]) to generate test cases independent from the original test suite and Xin *et al.* [53] proposed to generate tests that are ad-hoc to cover the syntactic differences between the generated patch and the ground-truth (i.e., developer-provided patch). A patch is considered as overfitting if it fails any of the newly-generated test cases. Xiong *et al.* [54] on the other hand assumes that the ground-truth is missing. They therefore proposed PATCH-SIM, which builds on the similarities among test execution traces for predicting patch correctness. The basic idea is that a patch is likely to be correct if execution traces of the patched program on previously failing tests are significantly different from those of the buggy program.

In other directions, Yang *et al.* [56] proposed a heuristic where a patch is considered correct if its inferred invariants are identical to those of the ground-truth. Wang *et al.* [52] performed a systematic study and found that combining dynamic approaches with static features achieves promising results and is thus a potential direction.

Overall, prior works in patch assessment have generally exclusively focused on the syntax and semantics of the generated patch, and to some extent of the patched program behavior. We observe that the literature has largely overlooked the characteristics of the bug itself. In this work, we introduce a novel perspective of patch assessment where the correlation between the patch and the bug is investigated. Our intuition is thus that the description of a correct patch for a specific bug has a latent correlation with that bug's description. By leveraging the bug reports as an informal description of the buggy behavior and a natural language description of the patch, we propose an NLP-driven approach, Quatrain, which is capable to discriminate correct patches from incorrect ones.

***Leveraging NLP in program repair.*** Given that the target of program repair is to transform a buggy program into its correct version, a number of recent studies have considered it as a translation task. Building on the software naturalness hypothesis [14], researchers proposed to apply existing neural machine translation (NMT) techniques generally leveraged in natural language processing. Chen *et al.* [5] proposed a recurrent neural network (RNN) based approach that fixes one-line bugs by translating the buggy line into the correct line. Tufano *et al.* [50] designed another RNN based model that works at the method level: the model takes a buggy method as input and generates the entire fixed method as output. Lutellier *et al.* [32] proposed to separately encode the buggy line and its surrounding contexts (i.e., statements that appear before or after the buggy line). CURE [19], a more advanced approach, leverages pre-training techniques to help the model gain knowledge about the rigorous syntax of programming languages and how developers write code. Another recent study [34] investigated the feasibility of applying a large-scale pre-trained model, CodeBERT, to generate patches.

Overall, while these previous works apply NLP techniques to the patch generation process, our work investigates NLP models for patch correctness assessment.

***Leveraging bug reports in software engineering tasks:*** Bug reports are considered as invaluable resources for debugging activities since they typically contain detailed descriptions about the program failures as well as the clues of the fault (usually in the form of stack traces) [4]. A number of studies have exploited bug reports to facilitate diverse software engineering tasks. Liu *et al.* [27] and Koyuncu *et al.* [24] investigated the feasibility of building program repair systems based on bug reports, instead of the traditional test cases. Indeed, the primary motivation of their works is that the required test case in APR for triggering the bug may not be readily available in practice when the bug is reported. Fazzini *et al.* [10] and Zhao *et al.* [66] explored how to automatically reproduce program failures from bug reports without human intervention. By leveraging code change patterns mined from bug reports, Khanfir *et al.* [22] proposed an approach that injects realistic faults to improve mutation testing. Besides, bug reports have been utilized for constructing high-quality defect benchmarks for software testing [23, 43]. In our study, we leverage bug reports to model the semantics of the bug and thus better assess the patch correctness.

## 2.2 Hypothesis Validation

Our hypothesis is that there is a semantic correlation between a bug description and the associated (correct) patch description. To validate the existence of such a correlation, we conduct a preliminary experiment on a collected dataset. The experiment investigates the semantic similarity between the descriptions. To that end, we consider a ground truth dataset of Defects4J bugs for which a bug report is available and the commit messages describing developer-written patches are provided. These are denoted "original pairs". Then, we assign a randomly selected commit message to each bug report in order to build 'random pairs'. Finally, to capture the semantics of the natural language sentences forming the descriptions (of bugs and patches), we utilize a pre-trained deep learning model BERT [7] (introduced in Section 3.4), which embeds the descriptions into vectors. We standardize[1] these vector values to eliminate the influence of dimension on the similarity computation. Finally, we calculate Euclidean distance for all pairs. Figure 2 presents the distribution for original pairs and random pairs. The results show that the original (i.e., ground truth) bug report and associated commit message pairs are more similar than random pairs. The Mann–Whitney–Wilcoxon test [35] (p-value: 1.2e-32) further validates the significance of the distribution difference. Note that we use semantic similarity as a metric to determine correlation. The validation of the existence of such correlation motivates the Quatrain approach, where NLP modelling is leveraged to develop a classification approach of patch correctness by building on predicting the relevance of a patch (based on its description) for a bug (given its description).

[1] In a standardized dataset, each feature has a range of values with a mean of 0 and standard deviation of 1.

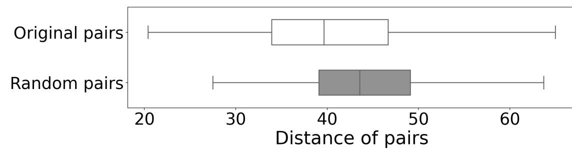


**Figure 2: Distributions of Euclidean distances between bug and patch descriptions.**

# 3 APPROACH

In this section, we first describe the overview of our proposed approach. Then, we fill in the details of the approach with specific steps separated in several subsections.

**[Overview]:** The intuition we build on is that the natural language description of a bug-fixing patch is semantically related to the bug report describing that specific bug: the bug report *describes the problem (bug)* and the patch description *describes the solution to the problem.* This semantic relation between a bug report and its associated patch is similar to the QA relation between questions and their answers in NLP. We present the definition as follow:

> *Definition 3.1 (Patch Correctness Prediction as a QA Problem).* Given a bug report in natural language $br_{nl}$, a patch $patch_c$ for the reported bug, and a natural language description $patch_{nl}$ of the patch, predict whether the QA-like pair $(br_{nl}, patch_{nl})$ is matching or not. I.e, predict whether the patch $patch_{nl}$ is relevant to (*answers*) the bug report $br_{nl}$ (*the question*) or not.

To solve this problem, we propose an approach, Quatrain, which takes as input a program whose buggy behavior is described in a bug report and the associated patch generated by an APR tool, and outputs a prediction of correctness. Figure 3 provides an overview of the approach, which includes two phases: training (offline) and prediction (online).

In the training phase, given a batch of a buggy program, Quatrain first extracts the bug NL description (bug report) from the program repository in an automatic way; then, for each candidate patch associated to the bug, it generates the patch NL description by leveraging a code change summarization tool (e.g., a commit message generator - cf. Sections 3.1 and 3.2). Subsequently, Quatrain requires a large number of positive (i.e. correct) and negative (i.e. incorrect) examples to train a classifier that predicts the correlation between a bug report and a patch description. Our third step (Section 3.3) thus focuses on building a dataset of positive and negative examples of pairs of bug reports and associated (in)correct patches. In the fourth step (Section 3.4), Quatrain converts the patch descriptions and the bug reports into vectors in high-dimensional vector space to enable model learning. Finally, in the fifth step (Section 3.5), Quatrain trains a neural QA classifier on the pairs of bug reports and patch descriptions.

In the prediction phase, Quatrain pre-processes a new buggy program and its associated candidate patch by applying the first, second, and fourth steps in Figure 3. It then uses the trained QA classifier to predict whether the candidate patch indeed answers the problem in the bug report. The answer is equivalent to a statement on the correctness of the plausible patch (yes:correct; no:incorrect).

## 3.1 Extraction of Bug Reports

The first step of our approach is to obtain descriptions of the bugs. A natural choice for finding such descriptions is to leverage bug reports. They exist in large numbers across projects and provide a NL description of program buggy behavior which, at least, describes the symptom of the bug. Bug reports are submitted via different platforms, e.g., issue trackers such as Jira and issues in GitHub. For our purpose, we use a script to automatically mine bug reports for the bug datasets that we use.

An official bug report typically includes three parts: title, description, and comments. In the benchmark that we build, some bug reports include comments where the correct solution or even the entire patch is posted. Note, however, that in our experimental assessment, we must assume that the bug has not yet been fixed. Thus, to remain practical and reduce bias, we discard all comments and leverage only the title and the description body of bug reports.

## 3.2 Generation of Patch Description

The second step of our approach is to summarize an APR-generated patch in natural language so as to obtain a semantic explanation of the changes applied in the patch. The idea here is to get a representation of the patch that is as close as possible to how a bug report describes, in natural language, what is the bug. If the patch is written by a developer (e.g., positive example patches in our training set), its associated commit message could be used as a proxy for such NL description of the patch. We use a script to mine the commit messages from developer repositories such as GitHub and collect the descriptions associated to the patches in our datasets.

Note however that commit messages are obviously not available for APR-generated patches. Therefore, we automatically generate patch descriptions with the help of state-of-the-art commit message generation techniques. In particular, we consider CodeTrans [9], an encoder-decoder transformer based model that has been developed to tackle several software engineering tasks. Quatrain uses CodeTrans-TF-Large, the largest such model which achieves the highest BLEU score so far of 44.41 on the commit message generation task.

During training, we obtain patch descriptions either from: (i) Manually written commit messages of bug-fixing patches provided by developers, or (ii) Automatically generated descriptions using CodeTrans for APR-generated patches.

## 3.3 Construction of Training Examples

Recall from Definition 3.1 that we are addressing a binary classification problem. To train a binary classifier, one needs to collect positive (i.e. correct) as well as negative (i.e. incorrect) examples. Therefore, the third step of Quatrain is to build a dataset of positive and negative training examples.

At a high level, a positive (or negative) training (or testing) example consists of a bug report and its associated patch.

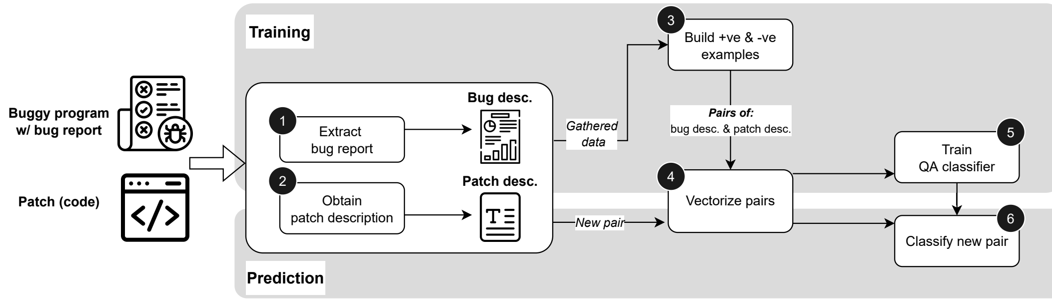


**Figure 3: Overview of the approach.**

*Definiton 3.2 (Bug report-patch description pair).* A *bug report-patch description* pair is a tuple $(br_{nl}, patch_{nl})$ of a bug report $br_{nl}$ and a patch description $patch_{nl}$ (in NL) of a *patch* that is intended to fix the bug reported in *br*.

*3.3.1 Positive Examples.* We collect two kinds of positive (correct) training examples. The first kind of correct examples are developer-written patches and their associated bug reports. The second kind of correct examples are APR-generated patches and their associated bug reports where the APR patches have been manually labeled in previous studies [25, 28, 47, 60]

*3.3.2 Negative Examples.* We need to create negative examples to train Quatrain to identify incorrect patches. To do so, we build two kinds of incorrect examples.

For the first kind of negative examples, we randomly assign developer-written patches to irrelevant bug reports. For example, we create a training sample by assigning the patch for bug-*x* with the bug report of bug-*y*. The rationale for creating this kind of negative examples is to mobilize the model to learn the hidden relations between bug reports and their associated patch descriptions. A patch that tackles a totally irrelevant bug would carry much less - if any - relation to the bug under examination.

The second kind of negative examples is created by selecting APR-generated patches that have been labeled as incorrect in previous studies [25, 28, 47, 60] and their associated bug reports. The idea is that those APR-generated patches were intended to address the specific bug under examination, but a manual verification revealed that they were incorrect. Correspondingly, the patch description generated for such incorrect patch does not correctly answer the bug report and thus could be considered as negative example.

## 3.4 Embedding of Bug Reports and Patches

To efficiently learn the relationship between bug reports and patch descriptions, we first need to convert the text into a numerical representations. Though there exist various techniques [3, 6, 20, 40] for transforming texts into numerical vectors, selecting the proper embedding technique is crucial, as it influences how precisely the vectors represent the text. Compared with popular embedding models such as Word2Vec [36], which uses a fixed representation for each word regardless of the context within which the word appears, BERT [7] has more advantages for representing texts: it produces word representations that are dynamically informed by the words around them. Thus, we employ BERT [7] as our initial embedding model for both bug reports and patch tokenized texts. The used model is a pre-trained large model with 24 layers and 1,024 embedding dimension trained on cased English text. After representing texts into a vector space, we can perform numerical computations on them, e.g., compute text similarity or correlation metrics.

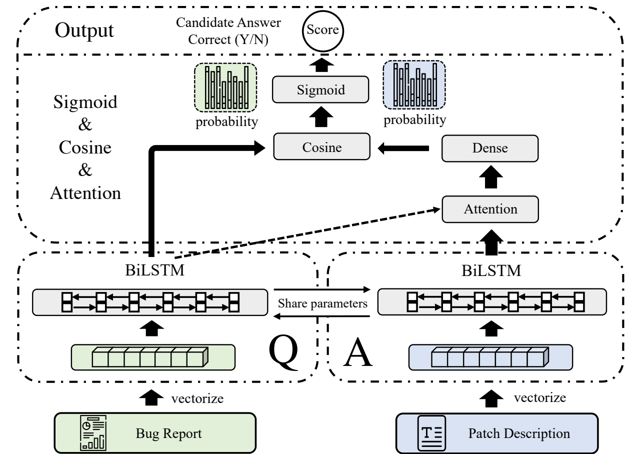


**Figure 4: Architecture of the neural QA model.**

## 3.5 Training of the Neural QA-Model

QA-Models are widely applied in Natural Language Processing (NLP) and Software Engineering (SE) communities. Existing literature on QA has addressed various tasks [12, 37, 42], among which, the task of answer selection shares fundamental similarities with our bug report-patch description matching problem, i.e., select a correct answer (patch) for a question (bug report). Tan *et al.* [45]s' approach exhibits powerful performance on this task by extending basic bidirectional long short-term memory (BiLSTM) model with an efficient attention mechanism. Thus, in the fifth step of Quatrain, we propose to adapt the extended QA model from Tan *et al.* to learn the correlation between bug reports and patch descriptions. We present the architecture of our adapted QA model in Figure 4.

The QA model requires two inputs: bug report and patch description, in their vectorized format (as per Section 3.4). Then, the BiLSTM layer takes the inputs to learn the correlation between the bug report and the patch description. Assuming input vectors are $vector_b$ and $vector_c$, we present the BiLSTM in Equation 1.

$$e_b = BiLSTM(vector_b) = [xb_1, xb_2, ..., xb_N] \in \mathcal{R}^{N\times dim}$$
$$e_c = BiLSTM(vector_c) = [xc_1, xc_2, ..., xc_N] \in \mathcal{R}^{N\times dim} \quad (1)$$

where $e_b$ and $e_c$ represent BiLSTM embeddings of one bug report $b$ and one associated patch description $c$. $N$ is the length of the input sequence and $dim$ refers to the dimension size of each sequence. $xb_i$ and $xc_j$ are the embeddings of i-th word in $b$ and $c$.

To better distinguish the correct patch from other patches based on the bug report, we employ an attention mechanism on the patch description to combine the most relevant information according to the bug report, similar to Tan *et al.* [45].

To this end, for each word embedding $xc_j$ in patch description, we compute the matrix product $e_b(xc_j)^T$. We then propagate the resulting vector through a softmax operator to obtain the impact weight $\alpha_{xc_j}$ of each word of bug report to $xc_j$.

$$\alpha_{xc_j} = Softmax(e_b(xc_j)^T) \in \mathcal{R}^N \quad (2)$$

where Softmax($x$) = $\frac{exp(x)}{\Sigma_i exp(x)}$, and *exp(x)* is the element-wise exponentiation of the vector $x$. Afterwards, we map the impact weight $\alpha_{xc_j}$ back to each bug report word embedding $xb_i$ to obtain attention representation $att_{xc_j}$,

$$att_{xc_j} = \Sigma_n^N \alpha_{xc_j,n} xb_i \in \mathcal{R}^{dim} \quad (3)$$

where $\alpha_{xc_j,n}$ means the *n-th* value of $\alpha_{xc_j}$. Then, we flatten $e_b$ and $att_{xc_j}$ to one-dimensional vectors $re_b$ and $re_c$ representing the bug report and patch description (with attention), respectively.

Finally, we compute cosine similarity between bug report vector $re_b$ and associated patch description vector $re_c$ and use the sigmoid activation function to normalize the output value of cosine layer to the value range of 0 and 1.

$$Score = Sigmoid(cosine(re_b, re_c)) \quad (4)$$

where Sigmoid(x) = $\frac{1}{1+exp(-x)}$. The *Score* is the prediction probability of patch correctness.

**[Hyper-parameters]:** The employed QA model is mainly based on BiLSTM. We set the max sequence length to 64 and the hidden state dimension size to 16 for the BiLSTM layer. During the training period, we iterate the model parameters by using an Adam optimizer with a leaning rate of 0.01. Considering the data size, we execute 10 training epochs to ensure the convergence of the model. The batch size at each epoch is 128.

## 3.6 Classifying a Pair of Bug Report and Patch

For a given buggy program and its APR-generated patch, QUATRAIN classifies the pair as being correlated or not by first extracting the bug description, generating a textual description of the patch, vectorizing the pair of texts, and finally querying the trained QA model. A prediction probability is a value between 0 (incorrect) and 1 (correct). QUATRAIN labels a patch as being correct or not based on a threshold on the prediction output (Section 5.1).

# 4 EXPERIMENTAL SETUP

We first enumerate the research questions that we investigate to assess the effectiveness of our approach. Then, we describe the dataset used for answering the questions. Finally, we present the evaluation metrics used in our study.

**Table 1: Datasets of labeled patches.**

| Benchmark | Subjects | Correct | Incorrect | All |
|---|---|---|---|---|
| Defects4J [21] | Tian *et al.* [47] | 1,344 | 1,017 | 2,361 |
| | Ye *et al.*[60] | 0 | 5,493 | 5,493 |
| | AVATAR[28], DLFix[25] | 59 | 38 | 97 |
| Bugs.jar [43] | Ye *et al.*[60] | 930 | 2,254 | 3,184 |
| Bears [33] | | 251 | 531 | 782 |
| **Total** | | 2,584 | 9,333 | 11,917 |
| **Total (deduplicated)** | | 2,260 | 9,092 | 11,352 |
| **Total (experiment)** | | 1,591 | 7,544 | 9,135 |

## 4.1 Research Questions

- **RQ-1:** *What is the effectiveness of QUATRAIN in patch correctness identification based on correlating bug and patch descriptions?* We evaluate QUATRAIN on a large dataset consisting of ground truth correct and incorrect patches.
- **RQ-2:** *To what extent does the quality of the bug report and of the patch description influence the effectiveness of QUATRAIN?* We perform two separate experiments: in the first, we consider the size of texts (i.e., number of words) as a proxy for quality, and we investigate whether there is a difference in quality measurement across correct and incorrect predictions. In the second experiment, given the original bug report and developer patch description pairs, we replace them alternatively with a random bug report or a tool-generated patch description and observe changes in performance measurements.
- **RQ-3:** *How does QUATRAIN perform in comparison with the state-of-the-art techniques for patch correctness identification?* We propose to compare our approach against static and dynamic approaches proposed in the literature for APR patch assessment.

## 4.2 Datasets

In this paper, we leverage benchmarks that are widely used in the program repair community and on which several APR tools have been applied to generate a large number of patches: Defects4J [21], Bugs.jar [43] and Bears [33]. Table 1 summarizes the patch dataset that we use for our experiments. First, we mainly collect the labeled patches (including developer patches) from the studies of Tian *et al.* [47] and Ye *et al.* [60]. We then supplement the dataset with the patches generated by AVATAR [28] and DLFix [25], which were not considered in these prior works. Considering that different APR tools may generate the same patches for the same bug, we use a simple string-based comparison script to deduplicate our patch dataset. Overall, we obtain a large duplicated patch assessment dataset of 11,352 patches consisting of 2,260 correct and 9,092 incorrect patches. Nevertheless, although we removed some duplicated patches, there are some semantically equivalent patches that could not be detected with our script. For instance, the two conditional statements '`if (dataset == null)`' and '`if ((dataset) == null)`' in Java are equivalent, although the extra parentheses make their raw strings mismatch. To reduce the bias of these duplications in our experiments, we design a specific dataset split scheme in Section 5.1.

**In our experiment:** We recall that our approach relies on measuring the correlation between the bug report (BR) and the patch description to predict patch correctness. The collected patches above

involve 1,932 unique bugs To obtain the associated bug reports, we mined their code repositories. Unfortunately, 631 bugs do not contain associated bug reports. Eventually, we were able to leverage 1,301 bug reports. Finally, for the 1,301 unique bugs, we obtain 9,135 available patches consisting of 1,591 correct and 7,544 incorrect patches for our experimental evaluation.

### 4.3 Metrics

Our objective in patch correctness identification is to recall as many correct patches while filtering out as many incorrect patches as possible. Thus, we follow the definitions of **Recall** proposed by Tian *et al.* for the evaluation of their BATS [47]:

- **+Recall** measures to what extent correct patches are identified, i.e., the percentage of correct patches that are identified from all correct patches.
- **-Recall** measures to what extent incorrect patches are filtered out, i.e., the percentage of incorrect patches that are filtered out from all incorrect patches.

$$+Recall = \frac{TP}{TP + FN} \tag{5}$$

$$-Recall = \frac{TN}{TN + FP} \tag{6}$$

where $TP$ represents true positive, $FN$ represents false negative, $FP$ represents false positive, $TN$ represents true negative.

**Area Under Curve (AUC) and F1**. We construct a deep learning-based NLP classifier to identify the correctness of the patch. Therefore, we use the two most common metrics, AUC and F1 score (harmonic mean between precision and recall for identifying correct patches), to evaluate the overall performance of our approach [16] .

## 5 EXPERIMENTS & RESULTS

We conduct several experiments to answer our research questions. In Sections 5.1 and 5.2, we focus on the evaluation of approach performance and analysis of approach input to performance. In Section 5.3, we compare against the state-of-the-art approaches.

### 5.1 Effectiveness of Quatrain

**[Experiment Goal]:** We answer RQ-1 by investigating to what extent the Quatrain approach, which predicts patch correctness by correlating bug and patch descriptions, is effective.

**[Experiment Design]:** In the literature, ML-based approaches to patch correctness identification are commonly evaluated using 10-fold cross validation (i.e., patch set is divided into 90% for training and 10% for test) [48]. However, as we noted in the analysis of our datasets, there are semantically equivalent patches. Thus the training and testing set may contain duplicate samples, which could lead to biased [2] experimental results due to data leakage (i.e., the model already sees some same test samples in the training phase).

Given the challenge to fully deduplicate the dataset, we propose to limit the bias via a new split scheme, referred to as *10-group cross validation*. A first manual analysis has shown that the duplicated

[2]We discuss this threat in Section 6

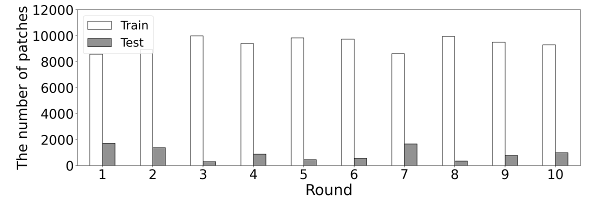


**Figure 5: Distribution of Patches in Train and Test Data.**

patches are typically generated by different APR tools while targeting the same buggy program. Therefore, we first randomly distribute 1,301 unique bugs (including 9,135 patches) into 10 groups: and every group contains unique bugs and their corresponding patches. Then, 9 groups are used as train data and the remaining one group is used as the test data. Finally, we repeat the selection of train and test groups for ten rounds and average the metrics obtained across the different experimental rounds. Through this 10-group cross validation scheme, each patch is able to be leveraged as train data and test data once, which fits the objective of cross-validation. Additionally though, during each train-test process, the unique bugs along with their sets of semantically equivalent patches are exclusively assigned to either train or test group. We trust that such a scheme will provide a realistic evaluation of the performance of learning-based approaches for patch correctness assessment.

Figure 5 shows the distribution of the number of patches assigned to train and test data at each round of 10-group cross validation. The overall ratio of train and test data splits is around 10:1. This ratio is close to typical 10-fold cross validation (9:1) and thus is appropriate to evaluate the performance of train-test based approaches.

**[Experiment Results]:** Using the presented 10-group cross validation, we provide the overall confusion matrix as well as the average +Recall (recall of correct patches) and -Recall (recall of incorrect patches) of Quatrain in Table 2.

Quatrain achieves high AUC at **0.886**, demonstrating the overall effectiveness of the QA model for patch correctness prediction. We note however that the F1 score (0.628) is relatively low. This metric is known to yield low values when the test data is imbalanced [18]: in our setting, the ratio is around 5:1 between the incorrect and the correct patch sets. We indeed confirm that that better F1 can be obtained by re-balancing the test data: with a ratio of 1:1 (1,591:1,591) at each round, the same trained classifier achieves a F1 score of **0.793**. Later, in our experiments, we mitigate the potential imbalance bias by comparing against state-of-the-art approaches on the same experimental settings (cf. Section 5.3). We found that +Recall and -Recall are sensitive to the selection of thresholds. When setting the threshold at a low value ( e.g., 0.1), we are able to identify all correct patches (+Recall=100%) but conversely none of the incorrect patches can be filtered out (-Recall=0%). Similarly, at the threshold value of 0.9, we filter out all incorrect patches but cannot recall any correct patch. Nonetheless, we see that Quatrain achieves promising results balanced between +Recall and -Recall when an adequate threshold is selected. For instance, Quatrain can recall 92.7% correct patches while filtering out 61.7% incorrect patches at a threshold value of 0.4 or +Recall of 73.9% and -Recall of 87.0% respectively at a threshold of 0.5. The results demonstrate our approach is effective on identifying correct and incorrect patches.

**Table 2: Confusion matrix of QUATRAIN prediction.**

| AUC | F1 | | Thresholds | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 |
| 0.886 | 0.628 | #TP | 1591 | 1582 | 1551 | 1475 | 1175 | 583 | 189 | 0 | 0 |
| | | #TN | 0 | 2388 | 3010 | 4653 | 6566 | 7261 | 7522 | 7544 | 7544 |
| | | #FP | 7544 | 5156 | 4534 | 2891 | 978 | 283 | 22 | 0 | 0 |
| | | #FN | 0 | 9 | 40 | 116 | 416 | 1008 | 1402 | 1591 | 1591 |
| | | +Recall(%) | 100 | 99.4 | 97.5 | 92.7 | 73.9 | 36.6 | 11.9 | 0 | 0 |
| | | -Recall(%) | 0 | 31.7 | 39.9 | 61.7 | 87.0 | 96.2 | 99.7 | 100 | 100 |

✍ **RQ-1** Experimental validation on our collected ground truth demonstrates the effectiveness of QUATRAIN in identifying correct patches and filtering out incorrect patches: our implementation achieves a +Recall of 92.7% and -Recall of 61.7% when the decision threshold is set at 0.4.

## 5.2 The Impact of Input Quality on QUATRAIN

**[Experiment Goal]:** QUATRAIN relies on specific steps to extract bug and patch descriptions once a patch candidate is generated to be applied for a buggy program. The quality of these descriptions may thus influence the performance of our approach. We investigate such an influence by attempting to answer three sub-questions:

- **RQ-2.1** *To what extent does the length of bug reports and patch descriptions influence the prediction performance?* We hypothesize that good descriptions should have more distinct words, and explore whether correct predictions are made on patch/bug descriptions of larger size.
- **RQ-2.2** *Does the NLP-based QA classifier actually correlate the bug report and the patch description?* We introduce noise in the test data and evaluate whether the classifier is actually looking at the correlation that we seek to check with the QA.
- **RQ-2.3** *Do generated patch descriptions provide the same learning value as developer-written commit messages?* We perform experiments where ground truth patch descriptions in the training set are alternatively switched between developer-written (assumed of high quality) and automatically generated (assumed of lower quality) commit messages.

**[Experiment Design (RQ-2.1)]:** We first define the length of input sentence as the number of distinct words included in the bug reports. Our assumption is that the presence of more distinct words in a textual description may indicate higher quality. Then, for each evaluation round of the 10-group cross validation scheme, we compute the boxplot distribution of length of bug and patch description for correct and incorrect predictions made by our model respectively. Finally, we calculate Mann Whitney Wilcoxon (MWW) to evaluate whether the difference of length is significant across the distributions. The analysis is made on both the length of bug report and patch description.

**[Experiment Results (RQ-2.1)]:** Figure 6 presents the distributions of patch description lengths for each round of prediction. We observe that, overall in most groups, the length of patch description are bigger in the correct predictions than in the incorrect predictions: the model is effective when the patch description has larger size. The Mann–Whitney–Wilcoxon test (p-value: 4.1e-16) further confirms that the difference of length is statistically significant. In

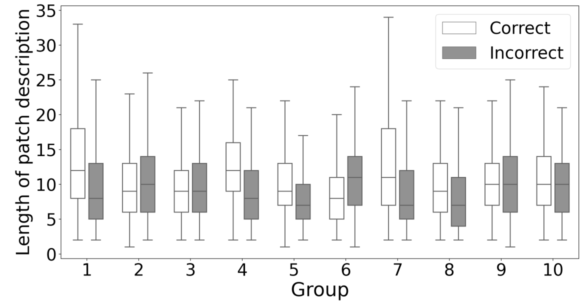


**Figure 6: Impact of length of patch description to prediction.**

contrast, the difference for the case of bug reports was not found to be statistically significant.

✍ **RQ-2.1** The higher the quality of patch description (i.e., in terms of text length), the more QUATRAIN is accurate in predicting patch correctness.

**[Experiment Design (RQ-2.2)]:** We recall that our NLP model is designed to correlate the bug report and patch description to predict the patch correctness. To validate that some correlation is indeed learned by the devised model, we investigate the influence of associating wrong bug reports to some patches in the test set. We consider the dataset of 1,301 developer-written patches in this experiment since the developer patch description and associated bug report are known to be indeed related by construction. We first compute the performance achieved by QUATRAIN in the prediction of correct patches. Then, for the patches that QUATRAIN correctly predicts (recall), we re-run the classification test where we replace the original bug reports with other randomly selected bug reports among the test data. We investigate whether this breakdown of the correlation between bug report and patch description is reflected in the prediction performance of NLP model.

**[Experiment Results (RQ-2.2)]:** Figure 7 presents the distribution of prediction probability of QUATRAIN for the 1,073 correct patches when the classifier is applied on the ground truth pairs (i.e., original pairs) and when the classifier is applied on pairs where the patch is associated to a random bug report (i.e., random pairs) . As we see from the boxplot, the lowest value of the distribution of original pairs (white box) is around 0.5. This is normal by construction: we set 0.5 as the threshold probability for deciding correctness, and our data is focused on cases where the prediction was correct. After breaking the correlation of bug report and patch description pairs, we found that QUATRAIN yields some prediction probability values smaller than 0.5 (i.e., they will be wrongly-classified as incorrect) although the patches are correct. The Mann–Whitney–Wilcoxon test (p-value:4.0e-35) confirms that the difference of median probability values is statistically significant between the two distributions. Concretely, 22% (241/1,073) of developer patches, which were previously predicted as correct, are no longer recalled by QUATRAIN after they have been associated to a random bug report. These results suggest that QUATRAIN indeed assesses the correlation between the bug report and the patch description for predicting correctness.

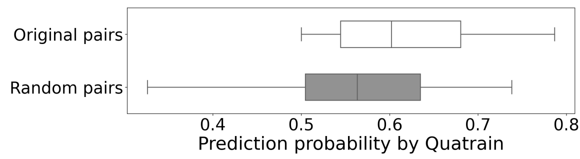


**Figure 7: The distribution of probability of patch correctness on original and random bug report.**

✍ **RQ-2.2** When developer patches are paired with random bug reports, Quatrain is no longer able to predict over 20% of them correctly. The results suggest that the QA learner in Quatrain indeed assesses the correlation between the bug report and the patch description for predicting correctness.

**[Experiment Design (RQ-2.3)]:** Commit messages are generally accepted as high-quality descriptions of changes since they are manually written by Developers. While CodeTrans is a state-of-the-art, its generated-descriptions should be lesser quality. Nevertheless, because developer-written commit messages are unavailable in practice for APR-generated patches, we must resort to automatic patch summarization tools such as CodeTrans. We evaluate the impact of the quality of patch description (developer-written vs. CodeTrans-generated) on the prediction performance. Our experiments focus on the developer patches only as in RQ-2.2. In the dataset, each patch has two kinds of descriptions, i.e., written by developer and generated by Codetrans. We first evaluate our approach based on developer-written descriptions. Then, we replace the developer descriptions with CodeTrans-generated descriptions to assess the performance evolution.

Besides, we speculate that Quatrain is more likely to correctly predict a correct patch if the generated description is similar to developer-written descriptions used in the training set, we conduct experiments to validate this hypothesis. Note however that the semantics of developer-written and generated descriptions should be equivalent as they describe the same developer patch. To measure the differences in the descriptions, we adopt the Levenshtein distance [3] and compute their textual similarity.

**[Experiment Results (RQ-2.3)]:** The experimental results show that Quatrain achieves a +Recall of 82% (1,073/1,301) when the input for test data uses developer-written descriptions of patches as in RQ-2.2. However, that metric (+Recall) drops by 37 percentage points to 45% when the developer-written descriptions are replaced with CodeTrans-generated descriptions. This demonstrates that the quality of patch description considerably impacts the prediction performance of Quatrain. Figure 8 displays the boxplot distribution of Levenshtein distance between developer description and generated description on correct and incorrect predictions respectively. In most of the groups, the white box (correct predictions) presents the shorter Levenshtein distance value, i.e., higher similarity. This result suggests that if a generated description has a quality that is as high as that of the developer description, Quatrain prediction ability will benefit from it. Finally, note that in Section 5.1, we evaluated Quatrain in a setting where all developer commit messages were replaced with generated descriptions: the AUC metric dropped by 11 percentage points to 0.774, confirming our findings.

[3]A classic metric for measuring the distance between two strings by calculating the minimal edit operations required.

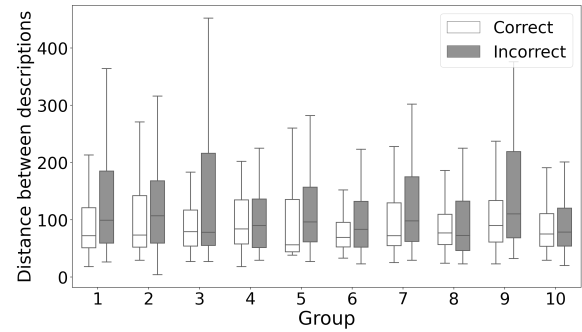


**Figure 8: Impact of distance between generated patch description to ground truth on prediction performance**

✍ **RQ-2.3** Patch descriptions generated by CodeTrans are often of different quality than ground-truth descriptions. Good patch descriptions help Quatrain identify more correct patches.

## 5.3 Comparison against the State-of-the-art

While previous RQs have shown that Quatrain is effective, in this section we compare it against state-of-the-art static and dynamic approaches. Finally, we investigate the complementarity of Quatrain to other existing approaches.

### 5.3.1 Comparing against Static Approaches.

We compare Quatrain against two state-of-the-art approaches: (i) A pure classification approach based on patch embeddings [48] and (ii) BATS which leverages the embedding of test cases to compute similarity among failing test cases and among associated patches [47].

**Quatrain vs. (supervised) DL-based Patch Classifier.** In Quatrain, we first leverage pre-trained Bert model to embed the natural language text of bug report and patch description of patch. Then, we build a deep learning classifier to capture the QA relationship between these descriptions to predict patch correctness. Since Tian *et al.*'s approach also use BERT and construct a classifier for patch correctness validation, we compare our approach against theirs. For a fair comparison, we reproduce their evaluation on our dataset. Concretely, when we train or test our model with divided-by-group patches, we consistently use the same patches for the training and testing of Tian *et al.*'s classifiers of Logistic Regression (LR) and Random Forrest (RF), following their experimental setup.

Table 3 presents the comparison results: Tian *et al.*'s best classifier (RF) achieves +Recall of 89.4% while filtering out 59.8% incorrect patches. Meanwhile, Quatrain achieves a better +Recall of 92.7%, and filters out slightly more incorrect patches (-Recall of 61.7%). Regarding the overall performance metrics AUC and F1, Quatrain outperforms the approach of Tian *et al.*. We finally investigate the complementarity of our approach. Among 9,135 patches, our approach identifies 7,842 patches, of which 2,735 patches cannot be identified by Tian *et al.*'s approach (RF).

Table 3: Quatrain vs a DL-based patch classifier [48].

| Classifier | Incorrect:Correct | AUC | F1 | +Recall | -Recall |
|---|---|---|---|---|---|
| Tian et al. (LR) | 7,544:1,591 (5:1) | 0.719 | 0.449 | 0.833 | 0.605 |
| Tian et al. (RF) | 7,544:1,591 (5:1) | 0.746 | 0.470 | 0.894 | 0.598 |
| Quatrain | 7,544:1,591 (5:1) | 0.886 | 0.628 | 0.927 | 0.617 |

Table 4: Quatrain vs BATS [47].

| Classifier | Incorrect:Correct | AUC | F1 | +Recall | -Recall |
|---|---|---|---|---|---|
| BATS (cut-off: 0.0) | 4,930:385 (13:1) | 0.549 | 0.149 | 0.647 | 0.452 |
| Quatrain | 4,930:385 (13:1) | 0.824 | 0.350 | 0.803 | 0.662 |
| BATS (cut-off: 0.8) | 367:41 (9:1) | 0.620 | 0.235 | 0.805 | 0.436 |
| Quatrain | 367:41 (9:1) | 0.832 | 0.462 | 0.902 | 0.453 |

**Quatrain vs. (unsupervised) BATS.** BATS[47] is the most recent patch correctness assessment approach proposed by Tian *et al.*. It is devised based on a simple but novel hypothesis that when different defective programs fail to pass similar test cases, it's likely that the programs can be repaired by similar code changes. Given a buggy program, failing test cases and a plausible patch, BATS first searches the most similar failing test cases from other oracle programs. Afterwards, the associated correct patches that fix these similar test cases are extracted to compute their similarity with the generated plausible patch. BATS labels the plausible patch as correct if that similarity is beyond an inferred threshold, otherwise it is predicted as incorrect.

According to the authors' open-source artifacts, BATS is currently able to be evaluated on Defects4J and is not adapted for Bears and Bugs.jar. We thus conduct the comparison on the benchmark of Defects4J. Although Tian *et al.* demonstrated that BATS shows promising results on identifying patch correctness, its scalability is limited due to the lack of enough test cases in the search space to compute similarity. They thus added a cut-off on the similarity computation of test cases to focus on a subset of patches where BATS is applicable. We follow their experimental setup to reproduce BATS evaluation on our dataset with the cut-off of 0.0 (non-specific scenario) and 0.8 (the best performance in their evaluation). We compare our approach on the same available dataset.

As shown in Table 4, the configuration of cut-off incurs the reduction of patch set that can be evaluated. Our approach comprehensively outperforms BATS whether they filter dissimilar programs or not (cut-off: 0.0 and 0.8). Note that BATS is not able to scale its performance, in this scenario, to the entire dataset due to the lack of similar test cases. In addition, in this scenario, 180 out of 345 patches are exclusively identified by Quatrain.

*5.3.2* **Comparing against a Dynamic Approach.** We consider a dynamic approach where execution traces are also leveraged in the prediction of correctness. PATCH-SIM [54] is a state-of- the-art tool for dynamic assessment of patch correctness: it compares test execution information before and after patching a buggy program. The hypothesis they proposed is that correct patches tend to change the behavior of execution of failing test cases and retain the behavior of passing test cases. Due to the failure of prediction for part of the patches [4] and limitation of timeout, we can apply PATCH-SIM to 3,546 patches. The results in Table 5 show that our approach filters out more incorrect patches while reaching same +Recall compared to PATCH-SIM. Most of the patches (1,856/3,149) that we identify are not correctly predicted by PATCH-SIM. Note that the low values of F1 score (both for PATCH-SIM and Quatrain) are due to the extremely imbalanced ratio of 44:1 in incorrect:correct sets.

Table 5: Quatrain vs (execution-based) PATCH-SIM [54].

| Classifier | Incorrect:Correct | AUC | F1 | +Recall | -Recall |
|---|---|---|---|---|---|
| PATCH-SIM | 3,468:78 (44:1) | 0.581 | 0.053 | 0.769 | 0.392 |
| Quatrain | 3,468:78 (44:1) | 0.792 | 0.127 | 0.769 | 0.667 |

✍ **RQ-3** Comparing against state-of-the-art static and dynamic approaches, Quatrain achieves competitive (or better) performance in predicting patch correctness.

## 6 DISCUSSION

We enumerate a few insights from our results and discuss the threats to the validity of our study.

### 6.1 Experimental Insights

[*Insufficient deduplication of semantically-equivalent patches may lead to biased prediction performance.*] As we mentioned in the experimental design in Section 5.1, the traditional 10-fold cross validation scheme may assign the same semantically-equivalent patches simultaneously into both train and test datasets. In practice, this setup violates the principles in machine/deep learning-based evaluations since it's equivalent to letting the models cheat by learning knowledge from test data during the training process[1, 17, 65]. To showcase this bias in the results, we propose to focus on a straightforward classifier using a random forest on the embeddings of the bug report and the patch: when using 10-fold cross validation scheme on our ground truth dataset, the achieved AUC is as high as 0.978 (with F1 at 0.860); however, when using our deduplication scheme (10-group cross validation based on bug ID), the AUC drops to 0.780 (and F1 at 0.344).

[*Generating high quality code change description can help identify patch correctness*] We found that the quality of code change description influences the prediction performance of Quatrain. In RQ-2.1 and RQ-2.3, the experimental results show the model makes more correct predictions when addressing longer or more developer written-similar code change description. Our experiments offer some evidence to encourage the community to design advanced patch summarization approaches. Quatrain indeed can become a prime candidate for leveraging such research output to further increase the practicality and adoption of automated program repair.

### 6.2 Case Study

Figure 9 presents an example correct patch generated by DLFix, an APR tool, for Defects4J bug Lang-7. Quatrain successfully predicts

[4] We reported the problem to the PATCH-SIM authors and we are still waiting for their response.

```
--- ./src/main/.../NumberUtils.java
+++ ./src/main/.../NumberUtils.java
@@ -449,9 +449,7 @@
         if (StringUtils.isBlank(str)) {
             throw new NumberFormatException("A blank string is
      not a valid number");
         }
-        if (str.startsWith("--")) {
-            return null;
-        }
+
         if (str.startsWith("0x") || str.startsWith("-0x") || str.
      startsWith("0X") || str.startsWith("-0X")) {
```

**Figure 9: A correct generated patch for Defects4J Lang-7.**

its correctness while BATS fails to do so. The associated bug [5] is reported as follows:

```
Title: NumberUtils#createNumber - bad behaviour
for leading "--".
Description: NumberUtils#createNumber checks for
a leading "--" in the string, and returns null if
found. This is documented ...
```

BATS assumes that similar buggy programs require similar patches to fix. To predict the generated patch correctness, BATS searches for a buggy program that fails on similar failing test cases with Lang-7. The retrieved program is the bug Lang-16 [6] in Defects4J. BATS predicts the generated patch is correct if it's similar with the developer patch addressing bug Lang-16. However, the retrieved bug Lang-16 is not related with bug Lang-7 even though they have similar test cases and require a dissimilar patch to fix. Thus, BATS fail to predict the generated patch correctness.

Consider however the NL description of the patch as it is generated by CodeTrans:

```
removed the unnecessary "" -- "" from NumberUtils .
startsWith ( ) , it was restricting our.
```

The syntactic and semantic correlation between the bug and patch description is obvious, which supports the fact that Quatrain predicts the patch as correct.

## 6.3 Threats to Validity

The implementation of Quatrain uses a pre-trained BERT to embed bug and patch descriptions before feeding them into the QA model. Quatrain also uses CodeTrans to generate patch descriptions. These choices may have influenced greatly our results. The associated threat is nevertheless limited since these constitute the state-of-the-art in their respective domains.

Our evaluation dataset includes 9,135 patches, though it is highly imbalanced (83% incorrect vs. 17% correct patches). This imbalance may bias our results. We mitigate this bias by stressing more on AUC metric, rather than F1 score and by performing comparison experiments against the state-of-the-art.

Our patch correctness labels have been manually decided in prior work [47]. The accuracy of the labels and the ground truth constitute a threat to validity [57], which is mitigated in part by our comparison against the state-of-the-art on the same datasets.

[5] https://issues.apache.org/jira/browse/LANG-822

[6] An upper-case hex bug in https://issues.apache.org/jira/browse/LANG-746

Our experimental evaluation does not perform any fine-tuning of the hyper-parameters of the QA model or even the initial BERT model used for embedding bug and patch descriptions. The yielded performance may thus not be representative of what can be achieved.

## 7 CONCLUSION

In this paper, we present a novel perspective to the patch correctness assessment problem in automated program repair. Given a plausible patch, which is validated by an imperfect oracle, the need for correctness identification is acute, as several studies have revealed that state-of-the-art repair tools generate overfitting patches. Our idea is that a correct patch is the one that answers to the problem revealed by the execution failure (bug). We therefore design Quatrain, a neural network architecture that leverages NLP to learn to correlate bugs and patch descriptions and produce a Question-Answering based classifier. Given a buggy program, we consider its bug report and leverage CodeTrans to generate descriptions for all APR-generated patches targeting the bug. Then, we use these NL descriptions of bugs and patches to feed the QA classifier of Quatrain. The classification decision serves as a prediction of patch correctness. The experimental results show that our approach identifies 92.7% correct patches and filter 61.7% incorrect patches with an AUC of 0.886. We then investigate and discuss the influence of the quality of the input (bug report and code change description) on the effectiveness of Quatrain. We also perform experiments to demonstrate that Quatrain indeed learns and builds on the correlation between the bug report and the patch to make the predictions. Finally, we reproduce recent state-of-the-art static and dynamic patch assessment tools on our dataset and show that Quatrain exhibits comparable or better effectiveness in recalling correct patches and filtering out incorrect patches. Insights from our work open new research direction in patch assessment, but also provide a novel use case for a large body of the literature that is focused on code summarization.

## ACKNOWLEDGMENTS

This work was supported by the NATURAL project, which has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant No. 949014). Kui Liu was also supported by the National Natural Science Foundation of China (Grant No. 62172214), the Natural Science Foundation of Jiangsu Province, China (Grant No. BK20210279), and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2020A06).

## REFERENCES

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

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[3] Prafulla Bafna, Dhanya Pramod, and Anagha Vaidya. 2016. Document clustering: TF-IDF approach. In *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*. IEEE, 61–66.

[4] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What makes a good bug report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 308–318.

[5] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.

[6] Kenneth Ward Church. 2017. Word2Vec. *Natural Language Engineering* 23, 1 (2017), 155–162.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 4171–4186. https://doi.org/10.18653/v1/n19-1423

[8] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 302–313. https://doi.org/10.1145/3338906.3338911

[9] Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. CodeTrans: Towards Cracking the Language of Silicon's Code Through Self-Supervised Deep Learning and High Performance Computing. *arXiv preprint arXiv:2104.02443* (2021).

[10] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 141–152.

[11] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011.*

[12] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Yuan-Fang Li. 2021. Code2Que: A tool for improving question titles from mined code snippets in stack overflow. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1525–1529.

[13] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2017), 34–67.

[14] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE, 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[15] Thong Hoang, Hong Jin Kang, Julia Lawall, and David Lo. 2020. CC2Vec: Distributed Representations of Code Changes. In *Proceedings of the 42nd International Conference on Software Engineering*. ACM, 518–529. https://doi.org/10.1145/3377811.3380361

[16] Mohammad Hossin and Md Nasir Sulaiman. 2015. A review on evaluation metrics for data classification evaluations. *International journal of data mining & knowledge management process* 5, 2 (2015), 1.

[17] Paul Irolla and Alexandre Dey. 2018. The duplication issue within the drebin dataset. *Journal of Computer Virology and Hacking Techniques* 14, 3 (2018), 245–249.

[18] László A Jeni, Jeffrey F Cohn, and Fernando De La Torre. 2013. Facing imbalanced data–recommendations for the use of performance metrics. In *2013 Humaine association conference on affective computing and intelligent interaction*. IEEE, 245–251.

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

[20] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. 2016. FastText.zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651* (2016).

[21] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 23rd International Symposium on Software Testing and Analysis*. ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[22] Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2020. Ibir: Bug report driven fault injection. *arXiv preprint arXiv:2012.06506* (2020).

[23] Misoo Kim, Youngkyoung Kim, and Eunseok Lee. 2021. Denchmark: A Bug Benchmark of Deep Learning-related Software. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 540–544.

[24] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: Bug Report driven Program Repair. In *Proceedings of the 27the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 314–325. https://doi.org/10.1145/3338906.3338935

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

[26] Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. 2022. Context-Aware Code Change Embedding for Better Patch Correctness Assessment. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 51 (may 2022), 29 pages. https://doi.org/10.1145/3505247

[27] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. 2013. R2Fix: Automatically generating bug fixes from bug reports. In *2013 IEEE Sixth international conference on software testing, verification and validation*. IEEE, 282–291.

[28] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. AVATAR: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 456–467. https://doi.org/10.1109/SANER.2019.8667970

[29] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2021. A Critical Review on the Evaluation of Automated Program Repair Systems. *Journal of Systems and Software* 171 (2021), 110817. https://doi.org/10.1016/j.jss.2020.110817

[30] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the Efficiency of Test Suite based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *Proceedings of the 42nd International Conference on Software Engineering*. ACM, 615–627. https://doi.org/10.1145/3377811.3380338

[31] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*. IEEE, 702–713. https://doi.org/10.1145/2884781.2884872

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

[33] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 468–478. https://doi.org/10.1109/SANER.2019.8667991

[34] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 505–509.

[35] Patrick E McKnight and Julius Najab. 2010. Mann-Whitney U Test. *The Corsini encyclopedia of psychology* (2010), 1–1.

[36] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[37] Sewon Min, Minjoon Seo, and Hannaneh Hajishirzi. 2017. Question answering through transfer learning from large fine-grained supervision data. *arXiv preprint arXiv:1702.02171* (2017).

[38] Martin Monperrus. 2020. The living review on automated program repair. (2020).

[39] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: feedback-directed random testing for Java. In *In OOPSLA '07 Companion*. ACM, 815–816.

[40] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[41] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 24th International Symposium on Software Testing and Analysis*. ACM, 24–36. https://doi.org/10.1145/2771783.2771791

[42] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[43] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. 2018. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th IEEE/ACM International Conference on Mining Software Repositories*. ACM, 10–13. https://doi.org/10.1145/3196398.3196473

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

[45] Ming Tan, Cicero dos Santos, Bing Xiang, and Bowen Zhou. 2015. Lstm-based deep learning models for non-factoid answer selection. *arXiv preprint arXiv:1511.04108* (2015).

[46] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 727–738.

[47] Haoye Tian, Yinghua Li, Weiguo Pian, Abdoul Kader Kabore, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. 2022. Predicting Patch Correctness Based on the Similarity of Failing Test Cases. *ACM Transactions on Software Engineering and Methodology* (2022). https://doi.org/10.1145/3511096

[48] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F Bissyandé. 2020. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 981–992. https://doi.org/10.1145/3324884.3416532

[49] Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F Bissyandé. 2022. The Best of Both Worlds: Combining Learned Embeddings with Engineered Features for Accurate Prediction of Correct Patches. *arXiv preprint arXiv:2203.08912* (2022).

[50] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology* 28, 4 (2019), 19:1–19:29. https://doi.org/10.1145/3340544

[51] Shangwen Wang, Ming Wen, Liqian Chen, Xin Yi, and Xiaoguang Mao. 2019. How Different Is It Between Machine-Generated and Developer-Provided Patches? An Empirical Study on The Correct Patches Generated by Automated Program Repair Techniques. In *Proceedings of the 13th International Symposium on Empirical Software Engineering and Measurement*. IEEE, 1–12. https://doi.org/10.1109/ESEM.2019.8870172

[52] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated Patch Correctness Assessment: How Far are We?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 968–980. https://doi.org/10.1145/3324884.3416590

[53] Qi Xin and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 226–236. https://doi.org/10.1145/3092703.3092718

[54] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 789–799. https://doi.org/10.1145/3180155.3180182

[55] Dapeng Yan, Kui Liu, Yuqing Niu, Li Li, Liu Zhe, Zhiming Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2022. Crex: Predicting patch correctness in automated repair of C programs through transfer learning of execution semantics. *Information and Software Technology* 107043 (2022). https://doi.org/10.1016/j.infsof.2022.107043

[56] Bo Yang and Jinqiu Yang. 2020. Exploring the Differences between Plausible and Correct Patches at Fine-Grained Level. In *Proceedings of the 2nd International Workshop on Intelligent Bug Fixing*. IEEE, 1–8. https://doi.org/10.1109/IBF50092.2020.9034821

[57] Deheng Yang, Yan Lei, Xiaoguang Mao, David Lo, Huan Xie, and Meng Yan. 2021. Is the Ground Truth Really Accurate? Dataset Purification for Automated Program Repair. In *2021 IEEE 28th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[58] Deheng Yang, Kui Liu, Dongsun Kim, Anil Koyuncu, Kisub Kim, Haoye Tian, Yan Lei, Xiaoguang Mao, Jacques Klein, and Tegawendé F Bissyandé. 2021. Where were the repair ingredients for Defects4j bugs? *Empirical Software Engineering* 26, 6 (2021), 1–33.

[59] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 831–841. https://doi.org/10.1145/3106237.3106274

[60] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering* (2021).

[61] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. https://doi.org/10.48550/ARXIV.2203.12755

[62] He Ye, Matias Martinez, and Martin Monperrus. 2019. Automated Patch Assessment for Program Repair at Scale. *CoRR* abs/1909.13694 (2019). http://arxiv.org/abs/1909.13694

[63] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-Based Backpropagation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1506–1518. https://doi.org/10.1145/3510003.3510222

[64] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2019. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system. *Empirical Software Engineering* 24, 1 (2019), 33–67. https://doi.org/10.1007/s10664-018-9619-4

[65] Yanjie Zhao, Li Li, Haoyu Wang, Haipeng Cai, Tegawendé F Bissyandé, Jacques Klein, and John Grundy. 2021. On the impact of sample duplication in machine-learning-based android malware detection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–38.

[66] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. 2019. Recdroid: automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 128–139.