Identifying Incorrect Patches in Program Repair Based on Meaning of Source Code

QUANG-NGOC PHUNG1, MISOO KIM2, and EUNSEOK LEE.3

1Dept. of Electrical and Computer Engineering, Sungkyunkwan University, Suwon, South Korea (e-mail: ngocpq@skku.edu)
2Institute of Software Convergence, Sungkyunkwan University, Suwon, South Korea (e-mail: misoo12@skku.edu)
3College of Computing, Sungkyunkwan University, Suwon, South Korea (e-mail: leees@skku.edu)

Corresponding author: Eunseok Lee (e-mail: leees@skku.edu).

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT)(2019R1A2C2006411)

ABSTRACT Automatic Program Repair (APR) techniques have shown the potential of reducing debugging costs while improving software quality by generating patches for fixing bugs automatically. However, they often generate many overfitting patches which pass only a specific test-suite but do not fix the bugs correctly.

This paper proposes MIPI, a novel approach to reducing the number of overfitting patches generated in the APR. We leverage recent advances in deep learning to exploit the similarity between the patched method’s name (which often encloses the developer’s intention about the code) and the semantic meaning of the method’s body (which represents the actual implemented behavior) for identifying and removing overfitting patches generated by APR tools.

Experiments with a large dataset of patches for QuixBugs and Defects4J programs show the promise of our approach. Specifically, in a total of 1,191 patches generated by 23 existing APR tools, MIPI successfully filters out 254 of the total 797 overfitting patches with a precision of 90% while preserving 93% of the correct patches. MIPI is more precise and less damaging to the APR than existing heuristic patch assessment techniques, achieving a higher recall than automated testing-based techniques that do not have access to the test oracle. In addition, MIPI is highly complementary to existing automated patch assessment techniques.

INDEX TERMS Automated program repair, APR, patch correctness, developer intentions, code2vec, bert

I. INTRODUCTION

Software is becoming ubiquitous in every aspect of our daily life, but unfortunately, they often contain bugs. Software bugs may cause not only serious financial loss but also toll on human life [1], [2]. However, fixing bugs in software is an extremely difficult and expensive task that may constitute up to 90% of software expenses. Automatic Program Repair (APR) techniques have shown the potential to reduce the costs while improving the quality of software by generating bug-fixing patches automatically [3]–[6]. APR techniques usually take a buggy program and a test suite containing at least one failing test case as input, and then try to modify the program to generate plausible patches that make the buggy program passes all test cases in the given test suite. Such patches, collectively called plausible patches, are then provided to developers as bug-fixing suggestions [5] or can be applied to fix bugs directly in production software [7].

Current APR methods have shown impressive results in generating many plausible patches for bugs in large real-world software projects [3], [4], [8]. However, recent studies have shown that a major portion of the plausible patches generated is incorrect, which is known as overfitting patches. These overfitting patches just make the program pass the provided test suite, but do not fix the bug correctly or they introduce new bugs into the program. Providing developers with many overfitting patches can degrade their debugging efficiency [9]–[11], and threaten the validity and the practical usability of APR.

To remove overfitting patches, existing approaches, [12]–[16], use automated test generation to enhance the provided test suite with automatically generated test cases. However, these approaches are computationally expensive and encountered with the test oracle problem to determine the test results
[12]. A study by Long et al. [17] suggests that leveraging additional information beyond the test suite is critical for isolating overfitting patches. Other approaches use additional information such as anti-patterns [18], program invariant [19], human-written patches [20], or static features of code [21]–[23] to heuristically assess patches correctness. More recent approaches [23]–[29] are mainly based on measuring the similarity between the patched code and the original code to rank patches such that the patches that are more similar to the original program are ranked first. These approaches rely on the assumption that the correct patch is more similar to the original program than other patches. However, several studies [30]–[32] showed that there are many correct patches that are less similar to the original program. In addition, APR tools often generate many plausible patches that modify the program at a non-buggy location [33], such patches are probably incorrect even though they are very similar to the original program. To alleviate these issues, we need to reflect the intention of the developers behind the original code itself.

In practical software development, developers are usually asked to use descriptive names when naming program identifiers (e.g., variables, methods, and types) for the readability and maintainability of their source code [34], [35]. These names often embed hints or summary information so that developers can guess the intended behavior without needing to read the implementation code in detail [35]–[37]. These embedded developer intentions provide valuable correctness information that is not expressed in test suites. However, to the best of our knowledge, there is a lack of existing automated patch assessment methods considering this valuable information.

In the last few years, deep learning has achieved impressive results in various natural languages as well as source code understanding tasks. From a large dataset collected from different sources on the Internet, they have trained models that can understand the meaning of identifier names and can automatically generate text descriptions for code snippets [38]–[42].

In this paper, we propose a novel approach to heuristically identify incorrect patches generated by APR tools. The core idea is to exploit the similarity between the developer intention (that developer encloses in the source code via natural text, such as JavaDoc comments or the name of program entities) and the semantic meaning of the implementing codes (which can be automatically identified using existing source code understanding models) to assess the correctness of patches. We realize the idea in MIPI (Meaning-based Incorrect Patches Identifier), an automated patch assessment tool. MIPI utilizes Code2Vec [39] model to identify the meaning of code snippets and employs the word embedding model BERT [43] to measure the meaning similarity between natural text descriptions. MIPI identifies the correctness of patches based on comparing the meaning similarity between the patched method’s name (which embeds the developer’s intention) and the semantic meaning of the original and patched method bodies (which represent actual implemented behavior).

The proposed approach MIPI has several advantages over the existing automated patch correctness assessment approaches. First, it does not need oracle information, which is required by automated testing and program invariant-based approaches, therefore, greatly simplifies its use in practice. This advantage comes from the novelty of exploiting the semantic meaning of the source code based on deep learning models. Second, it does not require determining a specific threshold to filtering patches, which is required by similarity-based techniques. This advantage comes from using the meaning distance from the original method to the developer given method name as a baseline for assessing the correctness of patched code.

We perform experiments with a dataset consisting of 1,191 patches generated by 23 APR tools and 817 patches written by developers for buggy programs in QuixBugs and Defects4J benchmarks. The experimental results show that the similarity between the meaning of the method name and its body can be exploited to assess the correctness of patches. MIPI successfully filters 254 out of 797 incorrect patches with the precision at 90% while preserving 93% of correct patches in the dataset of patches generated by APR tools. We compare MIPI with various existing automated patch assessment techniques based on heuristics (such as static code’s features similarity, behavior similarity, anti-patterns, and invariants), as well as deterministic techniques based on automated test generations. The results show that MIPI filters out more overfitting patches than automated test generation based techniques that do not have oracle information. MIPI achieves higher precision and preserves more correct patches than heuristic-based when filtering overfitting patches. In addition, MIPI is highly complementary to the existing techniques. Specifically, MIPI can filter out from 52 to 132 additional overfitting patches for each existing patch assessment technique.

We summarize our contribution in this paper as follows:

- To the best of our knowledge, there has been no study on the automated patch-correctness assessment that considers the developer’s intention embedded in the meaning of program entities’ names. We statistically found that the distance between the method name and correct patches is smaller than that between the method name and incorrect patches.
- We propose a novel patch correctness assessment technique that exploits the developer intention embedded in the method name. Our method does not require additional test cases and does not exploit noisy source codes that are not clearly determined whether they are faulty or clean.
- We validated the effectiveness of MIPI through experiments with a large dataset of patches and comparison to various existing automated patch assessment techniques.

The rest of this paper is organized as follows. We first discuss related work in Section II. Then we describe our
motivation in Section III. Section IV introduces our proposed method in detail. Section V describes our experiment procedures. Section VI shows the experimental results. Section VII-B discusses the threats to validity. The conclusions of our study are described in Section VIII.

II. RELATED WORK
A. AUTOMATIC PROGRAM REPAIR
Many automatic program repair techniques have been proposed by both academia and industry during the last decade [21], [23], [24], [44]–[54]. Most of them take a buggy program and a test suite as input and try to output a patch that makes the program pass all test cases in the given test suite. Their process typically consists of three main tasks: (1) fault localization, (2) patch generation, and (3) patch validation. First, they used automatic fault localization techniques [55]–[58] to identify modification points, which are a subset of possibly faulty program statements. The patch generation then modifies the program code at the identified modification points to generate variants of the program. Finally, the generated program variants are validated against the given test suite to check if they passed all of the test cases. Depending on how they generate the patch candidates, existing APR techniques can be classified into different categories, such as heuristic-search [46], [52], semantic-search [50], [53], or machine-learning-based APR [59].

B. OVERFITTING PATCHES AND EMPIRICAL STUDY ON PATCH CORRECTNESS
Qi et al. [60] conducted the first study investigating the correctness of patches generated by generate-and-validate (G&V) APR tools. They showed that most of the generated patches are incorrect, and most of them simply delete the functionality of the program. Smith et al. [10] and Motwani et al. [6] systematically evaluate the correctness of G&V APR tools by using independent test suites which are different from the test suite used for generating patches. Their evaluation results showed that 58.4% to 86.2% of the APR-generated patches are overfitting patches, which only pass the inputted test suite but not the independent test suite. These overfitting patches often break under-tested functionalities of the program and even make the patched program worse than the unpatched one. They also pointed out that larger test suites produce slightly higher-quality patches, but surprisingly, higher-coverage test suites correlate with a lower patch quality. In [61], Le et al. revisited the overfitting problem in semantic-based APR and found that the overfitting problem also occurs in semantic-based APR techniques.

Le et al. [62] conducted a user study for assessing the reliability of both manual and automated patch-correctness assessment methods. They showed that a manual assessment is subjective and more expensive, whereas automated test generation tools with oracles can identify only fewer than a fifth of the incorrect patches.

Long et al. [17] systematically analyzed the patch search spaces of the APR. They found that correct patches are sparse in the search spaces while the test-suite overfitting patches are typically orders of magnitude more abundant. Therefore, leveraging additional information beyond the test suite is critical for discriminating between overfitted and correct patches.

C. AUTOMATIC PATCHcorrectness ASSESSMENT
To identify the correctness of patches, a direct approach is based on the idea of using automated test generation techniques to enhance the provided test suite with automatically generated test cases [12], [13], [15], [16]. In studies by Ye et al. [63], [64], and Wang et al. [65], the authors utilized two popular automated test generation tools, Evosuite [66] and Randoop [67], to randomly generate new test cases for the buggy program for enhancing the given test suite. Xin et al. [15] proposed DiffTGen, a tool that identifies overfitting patches based on the test generation. They also used Evosuite to generate new test cases; however, the difference is that they focus on generating test cases that exploit the syntactic differences between the APR-generated patches and the human-patched code. One limitation of these approaches is that they require test oracles (e.g., a human or a correct program version) to determine the test results. Our approaches do not require oracle information to determine the correctness of the patches.

Opad [13] and SIM-PATCH [25] approaches also use automated test generation tools to generate new test cases for identifying overfitting patches; however, they do not require oracle information. Specifically, Opad is based on two predetermined test oracles (crash and memory safety) to identify overfitting patches. PATCH-SIM approximates the test results of a newly generated test case based on the result of the test case with the most similar test execution trace in the given test suite. They used the enhanced test suite to heuristically assess the correctness of the patches, considering that the passing tests might behave similarly whereas the failing test might behave differently on the buggy and patched programs.

Yang et al. [19] conducted a preliminary study on using dynamic invariants to discriminate between correct and incorrect patches. They showed that the majority of the APR-generated plausible patches lead to different runtime behaviors, compared to patches written by the developer.

As a difference between our approach and the above methods, our approach is based on a static analysis of the program code, whereas these approaches require executing test cases to identify the patch correctness.

Tan et al. [18] aimed to filter out overfitting patches using a set of predefined anti-patterns. These anti-patterns capture typical types of incorrect behaviors such as "append early exit", "delete CFG exit node", or "delete control statement".

Ye et al. [64] and Tian et al. [27] proposed training a model that can predict the correctness of patches. Ye et al. [64] extracted 4,199 static code features from the AST edit script between the original code and APR-patched code. From these extracted features, they learn an ensemble prob-
ablistic model for classifying and ranking APR-generated patches. Instead of relying on manually identified features, Tian et al. [27] and Lin et al. [29] investigated the benefit of using automatically learned embedding vectors of code changes as the feature vector for training a patch correctness classification model.

Similarity-based approaches [23]–[28] are based on a concept of similarity (such as syntactic similarity [23], [24], behavior similarity [25], [26], or embedding similarity [27]–[29]) to rank the APR-generated patches by their similarity to the original program. These approaches assume that the patch that is more similar to the original program is more likely to be correct. This assumption is reasonable in some sense because most of the current APR tools are focusing on simple bugs that the buggy program is slightly different from the correct program. However, this assumption may not hold in general; as shown by several studies, many patches that are less similar to the original program can also be correct [30]–[32], and many patches that are very similar to the original program are incorrect [30].

As different from the similarity-based approaches, our approach uses the developer’s intention enclosed in the meaning of descriptive code elements (e.g., method names), instead of the original code, as the origin coordinate for evaluating the correctness of patches. Our approach does not rely on the assumption that the original buggy program is very similar to the correct program, it can filter out overfitting patches that the similarity-based techniques are likely to keep (e.g., the patches that are similar to the original program but far away from the developer’s intention) and retain correct patches the similarity-based approaches are likely to filter out (e.g., the patches that are significantly different from the original program but close to the developer intention). In addition, because the meaning distance from the original code to developer intention represents the quality of the current code, the patches that are farther from the method name in comparison to the original code are likely to be worse. Thus, the score of the original code version can be a reasonable threshold for classifying correct and incorrect patches, relaxing the burden of selecting the threshold.

A. Nilizadeh et al. [68] used an automated verification tool to verify the correctness of the APR patched programs concerning a formal behavior specification written in JML. Their main goal is to evaluate the overfitting patches problem in APR. Their study showed that about 41% of APR-generated patches are overfitting patches. Using a verification tool can prove the correctness of a patch, however, a formal specification is often not available in practical software projects.

III. MOTIVATION

Although there are many APR techniques, most are quite ad-hoc. Because existing automated fault localization techniques are still inaccurate, APR often modified program code at both faulty and non-faulty locations. In addition, the patched code is often generated arbitrarily by replacing a possibly faulty code with a randomly selected new code, as long as it passes the test suite. As a result, they frequently generate patches that are nonsensical or significantly different from what the developer intends. Although these patches pass the test suite, they are likely to be overfitting patches.

Figure 1 shows an overfitting patch generated by an APR tool, Tbar [69], for bug Math6 in the Defects4J benchmark [70]. The patch modifies the method “int getIterations()” by replacing the variable iterations in the return statement with the variable evaluations. The statement in the original code line is bug-free; however, the fault localization in Tbar inaccurately identified it as a fault, and TBar replaced it with a new statement to generate the patch. Although the patch does not correctly fix the bug, it still passes the provided test suite, because the test suite is too weak.

The overfitting patch in Figure 1 is not filtered out by existing automated patch correctness assessment techniques. Automatic test generation-based techniques failed to find failing test cases for this patch because generating new test cases for a large and complex project such as Math is problematic. Similarity-based approaches consider the patch to be correct because the patched code is slightly different from the original code. The anti-patterns approach also considers the patch as correct because it does not contain any anti-patterns.

However, the developer may easily identify the patch in Figure 1 as incorrect without the need to run any test case, based on their understanding of the meaning of the method name as well as the source codes. Specifically, the method name indicates that the developer’s intention of the method is to "get the number of iterations". The meaning of the patched code version and the original code version is to "return a count of the evaluations" and to "return a count of the iterations", respectively. Because the meaning of the patched code is farther away from the developer’s intention of the method, compared to that of the original code, the patched code is worse than the original. Thus, it seems that the patch is nonsensical and likely to be an overfitting patch.

Recent researches in deep learning [38]–[43] show that, by learning from a large dataset obtained from various sources on the Internet, the computer is able to understand the semantic meaning of natural language text as well as source code elements. For example, natural language models, such as BERT [43], can understand that the word "queen" is close to the word "king", and the word "man" is close to "woman". Code understanding models, such as Code2Vec [39], show impressive results in predicting method names or generating text descriptions for code snippets across different projects. Motivated by these successes, we proposed leveraging recent
advances in deep learning to automatically identify incorrect patches in APR.

We use the examples in Figure 2 to illustrate the proposed method. Figure 2 shows the original buggy method \textit{bitcount} in the BitCount program in the QuixBugs benchmark and two different patched versions of the method. First, we extract the developer’s intention about the code from the method name, i.e., \textit{bit count}. We then use the Code2Vec model to predict possible method names for the original code and each patched code version. The method name Code2Vec predicted for the original buggy code as well as for the patched code 1 is \textit{"count bits"}, whereas its predicted name of the patched code 2 is \textit{"get count"}. Because the semantic meaning from the patched code 2, i.e. \textit{"get count"}, in comparison to the developer’s intention (i.e., \textit{"bit count"}) is far greater than that from the original buggy code (i.e., \textit{count bits}), the second patch is considered worse than the original buggy code and is more likely to be an incorrect patch.

IV. APPROACH

Our approach, MIPI (Meaning-based Incorrect Patch Identifier), for identifying the correctness of a patch.

A. OVERVIEW

The overall process of the proposed approach is shown in Figure 3. It takes a buggy program and a patch as input and consists of the following five main steps.

1) \textit{Code Information Extraction}. It analyzes the patch and extracts for each modified method, three pieces of information: (i) the original code snippet, (ii) the patched code snippet, and (iii) intention enclosed data that encloses the intention of the developer about the code.

2) \textit{Code Meaning Identification}. It uses a deep learning model to identify the meaning of the original and patched code snippets.

3) \textit{Developer Intention Extraction}. It analyzes the intention enclosed data to extract the developer’s intention information. The output of this step is a text sentence that expresses the intended behavior of the code.

4) \textit{Meaning Similarity Measurement}. It measures the semantic similarity between the extracted developer’s intention and the predicted meaning of the code snippets.

5) \textit{Patch correctness classification}. Based on the similarity, it produces a score that represents the correctness of the patches and heuristically filters out possible overfitting patches.

In the rest of this section, we describe each step in detail.

B. EXTRACTING CODE SNIPPETS INFORMATION

This step is performed by the \textit{Code snippet extractor} component in Figure 3. The goal of this step is to identify and extract information for code snippets that are modified by a given patch. For each modified code snippet, it extracts information including (1) a snippet of the corresponding patched code, and (3) intention enclosed data that contains information about the developer’s intention.

In general, we can work with code snippets at different meaningful granularity levels, such as code blocks, code slices, code segments, or methods. However, we focused only on the method level for two reasons. First, a method often implements a unit function; thus, its meaning is precise. Second, most existing deep learning models are trained on a dataset of methods; thus, they may perform best with code snippets at the method level.

In an ideal fault localization situation, where only the modified statements are buggy, the intention enclosed data can include information from all unmodified code elements. However, in real situations, the fault localization in APR is often inaccurate; consequently, the unmodified code might also contain bugs and, thus, might provide misleading information for extracting developer intention. To avoid possible misleading information, we excluded all code elements in the method body and includes only information from the class declaration (e.g., class name, inherited class, and interface names), method declaration (e.g., method name and method signature), and the technical document of the method into the intention enclosed data for extracting developer intention.

We implemented the \textit{Code Snippet Extractor} component based on the Gumtree-AST-Diff [72], a source code differentiating library. Because a patch may contain multiple modifications, each may occur in different methods. Thus, the output of this step may contain multiple triples, each triple for a modified method.

Hereafter, we use $P = \{m^1, \ldots, m^n\}$ to denote a patch, where $m^i$ denotes the $i$th modified method.

C. IDENTIFY THE MEANING OF CODE SNIPPETS

This step is performed by the \textit{Code Description Generator} component in Figure 3. The goal of this step is to predict the semantic meaning of a given code snippet.

We implement this step based on Code2Vec\textsuperscript{1} [39], which is a deep neural network for predicting the method name of a code snippet, and also represents a code snippet as a numeric vector. The core idea of Code2Vec is to represent a code snippet as a bag of path-contexts. Each path-context is a path connecting two leaf nodes in the Abstract Syntax Tree (AST) of the code snippet. The vector representation of a code snippet is a weighted aggregation of its path-contexts’ components, while the weight is learned during the training using the attention mechanism. The Code2Vec model was trained using 12,636,998 Java methods extracted from a dataset of 10,072 GitHub repositories provided in [73]. To be exact, the network was trained by using the method body to predict the name of the method.

The output of this step is a list of predicted method names that are most similar to the meaning of the input code snippet.

\textsuperscript{1}https://github.com/tech-srl/code2vec
D. EXTRACTING DEVELOPER INTENTION DESCRIPTION

The goal of this step is to generate natural text data that describes the intention of the developer about the code from the intention enclosed data extracted by the Code snippet extractor. In general, the developer’s intention regarding the implementation code can be embedded in various source code elements such as identifier names (e.g., variable, method, class, and interface), code comments, or JavaDoc documents. As the first attempt to use developer intention embedded in source code to identify patch correctness, we simply extract the developer intention description from the name of the modified method. Specifically, we split the method name into a sentence of words based on common naming conventions (based on Camel case, snake) and output the obtained sentence as a description of the developer’s intention. Future work might use more sophisticated approaches that also consider information from additional code elements.

E. MEASURING MEANING SIMILARITY

This step is performed by the Meaning Similarity Measurer components in Figure 3. Input a list of predicted possible meanings of a code snippet $B_{text} = \{b^1_{text}, ..., b^k_{text}\}$ and a text description of the developer intention, $intent_{text}$, the Meaning Similarity Measurer computes a score that shows their similarity.

First, it uses a word embedding model to represent the $intent_{text}$, and each $b^i_{text} \in B_{text}$ as a real-valued vector, called embedding vector, that encloses its meaning. A word embedding model [43] is a neural network model that is trained to represent words (and sentences) as vectors in a high-dimension vector space such that the words that have a more similar meaning are represented as closer points in the vector space. The meaning similarity between two text sentences can be measured by computing the distance between their embedding vectors.

The similarity score between the developer intention and the source code’s meaning is computed as the formula in Equation 1.

$$dist = \min_{\forall b^i_{text} \in B_{text}} \text{Distance}(intent_{vec}, b^i_{vec})$$ (1)

In the formula, $intent_{vec}$ denotes the embedding vector of $intent_{text}$, and $B_{vec} = \{b^1_{vec}, ..., b^k_{vec}\}$ denotes the embedding vectors of elements in $B_{text}$. Function Distance() is a function that returns the distance between two real-valued vectors. Intuitively, the meaning distance $dist$ is computed by measuring the distance from the vector $intent_{vec}$ to each
vector $v^{i}_{\text{vec}} \in B_{\text{vec}}$ and return the minimum distance value as the output score.

In our implementation, we use a pre-trained version of BERT [43] as the word embedding model. BERT is a language representation model that is a state-of-the-art natural network model for natural language understanding. The model was trained on a large corpus of natural language text from Wikipedia and BookCorpus. We compute the distance between two vectors using cosine distance.

**F. PATCH CORRECTNESS CLASSIFIER**

The goal of this step is to classify the correctness of the patch $P = \{m^1, ..., m^n\}$ based on the meaning similarity scores computed in the previous step.

We use $\text{dist}_{\text{org}}$ and $\text{dist}_{\text{pat}}$ to denote the distance from the developer’s intention to the meaning of the original code, and to the meaning of the patched code of the modified method $m^i$, respectively.

For each modified method $m^i$, the Patch correctness classifier first compute its similarity gain $m^i_{\text{gain}}$ as

$$m^i_{\text{gain}} = \text{dist}_{\text{org}} - \text{dist}_{\text{pat}}$$

A positive value of $m^i_{\text{gain}}$ means that the patched code version of method $m^i$ becomes better than its original code version, that is, the similarity to the developer’s intention is increased. By contrast, a negative value of $m^i_{\text{gain}}$ indicates that the patched code version of the method $m^i$ becomes worse than its original code version.

Based on the computed similarity gains, the Patch correctness classifier heuristically classifies the correctness for each modified method $m^i$ of the patch as follows:

$$m^i_{\text{class}} = \begin{cases} \text{incorrect} & \text{if } (m^i_{\text{gain}} < 0) \\ \text{correct} & \text{otherwise} \end{cases}$$

Specifically, if the modified method $m^i$ has a negative value of $m^i_{\text{gain}}$, then the method is classified as "incorrect". Otherwise, the modified method $m^i$ is classified as "correct".

The $m^i_{\text{gain}}$ received a negative value when $\text{dist}_{\text{pat}} > \text{dist}_{\text{org}}$, thus, the value of the $\text{dist}_{\text{org}}$ acts as a threshold for classifying the correctness of the patched code.

The value of $\text{dist}_{\text{pat}}$ is zero when the predicted meaning of the patched code version is exactly the same as the extracted developer intention description. This indicates that the patched code version of method $m^i$ is exactly or extremely close to the developer’s intended implementation code. In this case, the patched method $m^i$ will always be classified as "correct" because the condition $m^i_{\text{gain}} \geq 0$ is always true, regardless of the value of the $\text{dist}_{\text{org}}$.

The value of $\text{dist}_{\text{org}}$ is zero means that the predicted method name of the original code version is exactly the same as the developer’s given method name. This case happens when (1) the original code version is bug-free but the fault localization inaccurately determined it as buggy, or (2) the original code version is buggy but it is already very close to the correct version such that the Code2Vec model cannot distinguish them. In this case, only patches that have $\text{dist}_{\text{pat}} = 0$ are classified as "correct", and all others will be filtered.

We classify a patch as incorrect if at least one of its modified methods is classified as incorrect. Otherwise, the patch is classified as correct.

**V. EVALUATION**

**A. RESEARCH QUESTIONS**

- RQ1: Is the meaning of the method name more similar to the meaning of correct method bodies than to the meaning of buggy method bodies?
- RQ2: To what extent can the meaning distance from the original method body to the method name classify correct and incorrect APR patches?
- RQ3: How effective is MIPI in identifying overfitting patches?
- RQ4: How is MIPI compared to existing automated patch assessment techniques?
- RQ5: How is the performance of MIPI on APR patches generated with perfect fault localization?

The goal of RQ1 is to investigate whether the meaning similarity score between the method name and method body based on deep learning models can be a reliable means of assessing the correctness of patches.

The goal of the RQ2 is to investigate whether we can use the original method body as a threshold to classify correct and incorrect patches generated by the APR.

In RQ3, we focus on identifying the number of patches our approach filters out, among which, we focus on how many patches are correct and how many are incorrect. In addition, we discuss the reason why MIPI incorrectly filters out correct patches and does not filter overfitting patches.

In RQ4, we compared the effectiveness of MIPI with different existing automated patch-assessment techniques. It is emphasized by existing studies [62], [65] that there is no silver bullet technique for a patch correctness assessment; each technique (even manual analysis) has different strengths and drawbacks compared to each other. In this research question, we also investigate how our approach is complementary to existing techniques.

The goal of RQ5 is to investigate whether MIPI is still useful when a better or even perfect fault localization technique exists. There are two sources for overfitting patches in APR. One of them is because the fault localization technique points to the wrong locations. The other is because the APR generates an overfitting patch for the correct location. The first source can be eliminated by improving fault localization, and many recent fault localization techniques have been proposed (i.e., ProFL [74]). The second source is independent of the fault localization and remains even with a perfect fault localization technique.
B. DATASET

We collected a dataset containing both developer written and automatically generated patches for buggy programs in Defects4J [70] and QuixBugs [71] benchmarks, which are two widely used benchmarks in the APR research community.

We collected APR-generated patches for the QuixBugs benchmark from a recent study by H. Ye et al. [75]. It contains 338 patches generated by 11 different APR tools for fixing 16 out of a total of 40 buggy programs in the benchmark. We cannot obtain the patched code version for three patches generated by the Tibra tool for the POWERSET program because we found that the buggy program, in which the APR tool was applied to generate the patches, is different from the buggy program code provided in the benchmark. Thus, we remove these patches from the dataset. As a result, we obtained 335 APR-generated patches for 15 buggy programs in the QuixBugs benchmark.

We collect APR-generated patches for Defects4J from the evaluation package of Wang et al. [65]. This package contains 902 patches generated by 21 different APR tools for five open-source projects in the Defects4J benchmark, which is currently the largest set of APR patches for the Defects4J. The correctness of these patches was assessed by the authors of [75] and [65] using either an automatic tool or manual assessment. Because some bugs (such as Closure 63, 93) were deprecated in Defects4J version 2.0, we excluded all patches for these deprecated bugs from the dataset. We also discarded all patches that did not modify the body of any method (e.g., the patches that modify only the data field or constructor of classes), because the Code2Vec model, which we used to predict the meaning of code, accepts only the Java method as input. After filtering, we obtained 856 APR-generated patches for five projects in the Defects4J benchmark.

In summary, there are 206 bugs that can be patched by at least one APR tool in our dataset. Among them, 33% (67/206) of the bugs have only one patch, 67% (138/206) of the bugs have less than 5 patches and only 9.7% (20/206) of the bugs have more than 10 patches generated by all of the considered APR tools.

We collect developer written patches for each bug in the QuixBugs and Defects4J benchmarks by comparing the buggy code with the corresponding developer fixed version.

Overall, our dataset contains 2,008 patches, of which 1,191 are generated by APR tools and 817 patches are written by the developer. Table 1 list the details of the number of patches generated by the APR and developer for each benchmark in our dataset. Table 2 lists all 23 APR tools and the statistics of their generated patches for each benchmark in our dataset. Columns C, I, and 'Tot.' represent the number of correct patches, incorrect patches, and the total number of patches, respectively.

Each APR technique uses a different fault (or fix) localization technique. Among the 23 APR tools, Nopol [50] uses the value replacement technique [56] to identify potential fix locations. SequenceR [59] requires the fault locations to be given in the input. NPEFix [76] maintains a stack of exception handlers in runtime to determine buggy statements causing null pointer exceptions. Other APR tools use spectrum-based fault localization techniques [77], [78] for the fault localization step. Specifically, they often use GZolta [79], an off-the-shelf spectrum-based fault localization tool, with Ochiai metric [77] to obtain a ranked list of suspicious statements. Some techniques use additional techniques such as test case purification [80] to improve the fault localization accuracy, e.g., SimFix [21].

TABLE 1: Statistics of patches collected in our dataset.

| APR Tool | Patches | QuixBugs | Defects4J |
|----------|---------|----------|-----------|
|          | C      | I       | Tot.      | C      | I       | Tot.      |
| Arja [46] | 168    | 38    | 113 | 4      | 51    | 55      |
| AVATAR [81] | 51    | 0    | 0  | 17    | 34    | 51      |
| CapGen [23] | 65    | 0    | 0  | 25    | 40    | 65      |
| DynaMoth [82] | 24    | 1    | 1  | 2     | 1     | 2       |
| FixMiner [83] | 31    | 0    | 0  | 12    | 19    | 31      |
| GenProg [52] | 193   | 99   | 64 | 163   | 2     | 28      |
| Jaid [49] | 78    | 0    | 0  | 39    | 39    | 78      |
| JKali [3] | 22    | 0    | 0  | 3     | 19    | 22      |
| jMutRepair [3] | 23    | 2    | 1  | 4     | 16    | 20      |
| Kali [60] | 67    | 2    | 2  | 4     | 60    | 63      |
| kPAR [33] | 61    | 0    | 0  | 10    | 51    | 61      |
| SequenceR [59] | 64    | 0    | 0  | 17    | 47    | 64      |
| SimFix [21] | 62    | 0    | 0  | 21    | 41    | 62      |
| SketchFix [84] | 21    | 0    | 0  | 14    | 7     | 21      |
| SOFIFX [85] | 23    | 0    | 0  | 21    | 23    | 23      |
| TBar [69] | 66    | 0    | 0  | 22    | 44    | 66      |
| Cardumen [3] | 16    | 3    | 2  | 5     | 2     | 9       |
| RSRepair [86] | 71    | 6    | 25 | 31    | 2     | 38      |
| jGenProg [3] | 20    | 0    | 0  | 3     | 17    | 20      |
| Nopol [50] | 35    | 2    | 2  | 1     | 30    | 31      |
| ACS [54] | 20    | 0    | 0  | 13    | 7     | 20      |
| NPEFix [76] | 9     | 4    | 5  | 9     | 0     | 0       |
| Tibra [3] | 1     | 1    | 0  | 1     | 1     | 0       |
| Sum       | 1191  | 158  | 177 | 335   | 236   | 620     | 856     |

TABLE 2: Statistic of APR patches dataset

| APR Tool | Patches | QuixBugs | Defects4J |
|----------|---------|----------|-----------|
|          | C      | I       | Tot.      | C      | I       | Tot.      |
| Arja [46] | 168    | 38    | 113 | 4      | 51    | 55      |
| AVATAR [81] | 51    | 0    | 0  | 17    | 34    | 51      |
| CapGen [23] | 65    | 0    | 0  | 25    | 40    | 65      |
| DynaMoth [82] | 24    | 1    | 1  | 2     | 1     | 2       |
| FixMiner [83] | 31    | 0    | 0  | 12    | 19    | 31      |
| GenProg [52] | 193   | 99   | 64 | 163   | 2     | 28      |
| Jaid [49] | 78    | 0    | 0  | 39    | 39    | 78      |
| JKali [3] | 22    | 0    | 0  | 3     | 19    | 22      |
| jMutRepair [3] | 23    | 2    | 1  | 4     | 16    | 20      |
| Kali [60] | 67    | 2    | 2  | 4     | 60    | 63      |
| kPAR [33] | 61    | 0    | 0  | 10    | 51    | 61      |
| SequenceR [59] | 64    | 0    | 0  | 17    | 47    | 64      |
| SimFix [21] | 62    | 0    | 0  | 21    | 41    | 62      |
| SketchFix [84] | 21    | 0    | 0  | 14    | 7     | 21      |
| SOFIFX [85] | 23    | 0    | 0  | 21    | 23    | 23      |
| TBar [69] | 66    | 0    | 0  | 22    | 44    | 66      |
| Cardumen [3] | 16    | 3    | 2  | 5     | 2     | 9       |
| RSRepair [86] | 71    | 6    | 25 | 31    | 2     | 38      |
| jGenProg [3] | 20    | 0    | 0  | 3     | 17    | 20      |
| Nopol [50] | 35    | 2    | 2  | 1     | 30    | 31      |
| ACS [54] | 20    | 0    | 0  | 13    | 7     | 20      |
| NPEFix [76] | 9     | 4    | 5  | 9     | 0     | 0       |
| Tibra [3] | 1     | 1    | 0  | 1     | 1     | 0       |
| Sum       | 1191  | 158  | 177 | 335   | 236   | 620     | 856     |

C. METHODOLOGY

1) RQ1 and RQ2

We identified all modified methods for all patches in our dataset and grouped them into three different sets. The first set contains all methods that are modified by developer written patches. The second set contains all methods that are modified by APR-generated correct patches. The third set contains all methods that are modified by APR-generated incorrect patches. Table 3 shows the number of methods collected for each set. In Table 3, the column "Developer" shows the total of methods that are modified by developers. The columns "APR correct" and "APR incorrect" show the total number of methods that are modified by APR generated
To answer RQ1, we compared the distribution of the meaning distances from the buggy method body to the method name versus from the correct method body. We collected the dataset of buggy method bodies and their corresponding correct method bodies from the set of developer modified methods. Specifically, for each method modified by a developer written patch, the original method body is considered buggy and the patched method body is considered correct since developer patches are assumed to be correct in both fault location and fix generation.

To answer RQ2, we compared the distributions of the meaning distances from the original method body to the method name versus that from the APR correctly patched method body, and versus that from the APR incorrectly patched method body.

2) RQ3: How effective is our approach in identifying overfitting patches generated by APR tools?

We applied MIPI to classify the correctness of all APR-generated patches in our dataset. MIPI labels a patch "Incorrect" or "Correct" if it classifies the patch as incorrect or correct, respectively. We compared the classification results with the actual correctness label of the patches and counted the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) cases as follows:

- **True Positive (TP):** correctly classifying an overfitting patch as "Incorrect"
- **False Positive (FP):** wrongly classifying a correct patch as "Incorrect".
- **True Negative (TN):** correctly classifying a correct patch as "Correct".
- **False Negative (FN):** wrongly classifying an overfitting patch as "Correct".

We compare our method with existing automatic patch correctness assessment (APCA) techniques using the following metrics, which are well-known in statistics and medical diagnosis [28], [65]:

\[
\text{precision} = \frac{TP}{TP + FP} \quad (4)
\]

\[
\text{sensitivity} = \frac{TP}{TP + FN} \quad (5)
\]

\[
\text{specificity} = \frac{TN}{TN + FP} \quad (6)
\]

\[
\text{Destructive Ratio} = \frac{1 - \text{specificity}}{\text{sensitivity}} \quad (7)
\]

**Precision** (also called positive predictive value (PPV)) is the proportion of classified overfitting patches (by the APCA method) that are actually overfitting. Higher precision scores indicate a higher probability that a patch being filtered by the APCA is actually an overfitting patch. **Sensitivity** (also called recall or true positive rate (TPR)) is the proportion of overfitting patches (judged by the developer) that are classified as overfitting by the APCA method. **Specificity** (also called true negative rate (TNR)) is the proportion of correct patches (judged by a developer) that are classified as correct by the APCA method. Intuitively, sensitivity shows how many percentages of overfitting patches can be filtered out, whereas specificity shows how many percentages of correct patches can be retained by the APCA method. Higher sensitivity scores indicated that a larger proportion of overfitting patches could be filtered out. Higher specificity scores indicated that a larger proportion of correct patches could be retained, or a smaller proportion of correct patches were mistakenly filtered out.

Maximizing the sensitivity is interesting. However, it is a critical error if many correct patches have been mistakenly filtered out because the ultimate goal of APR tools is to generate the correct patches. In addition, an APR tool must have worked extremely hard to generate a correct patch, because correct patches are rare, whereas the search space of APR is often huge [17], [61]. Therefore, mistakenly filtering a correct patch may destroy all efforts of the APR tool.

To measure the trade-off between the risk (of mistakenly filtering out all correct patches) and the benefit (of successfully filtering out overfitting patches) when using an automated patch assessment technique, we propose a new metric called the **destructive ratio (DR)**, as shown in Equation (7). The destructiveness metric represents the ratio between the proportion of filtered correct patches (i.e., false-positive rate) and the proportion of filtered overfitting patches (i.e., true positive rate). The higher DR values indicate that the APCA technique might cause more destructive or negative effects on the APR tools. A destructive ratio of greater than 1 means that the proportion of the filtered correct patches is larger than the proportion of the filtered overfitting patches, that is, the risk outweighs the benefit of using the APCA. A destructive ratio of smaller than 1 means that the proportion of the filtered correct patches is smaller than the proportion of the filtered overfitting patches, that is, the risk is smaller than the benefit of using the APCA.

3) RQ4: Comparison to existing patch assessment techniques

Running existing automatic patch correctness assessment (APCA) techniques is extremely difficult and requires considerable time and resources. We used the same dataset of patches as the studies in [75] and [65], in which various APCA techniques are evaluated. Thus, we directly compare our results with the results of the existing techniques reported in these papers.

Specifically, for the QuixBugs dataset, we compared the

---

**TABLE 3: Dataset of modified methods.**

| Modified methods# | Developer | APR correct | APR Incorrect |
|-------------------|-----------|-------------|--------------|
| 1,263             | 388       | 861         |

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/
results of three APCA techniques reported by Ye et al. [75], i.e., Evosuite [66], InputSampling [87], and Invariant [19]. Evosuite and InputSampling are automated patch assessment techniques based on a ground truth program for automated test generation. Evosuite generates the test input using search-based test generation techniques, whereas InputSampling generates the test input by sampling the input space according to a specification. The invariant technique uses Daikon [88] to infer the program invariants from the ground truth version and then uses them to identify overfitting patches.

For the Defects4J dataset, we compared the results of 11 automated patch assessment techniques, which were reported in the study by Wang et.al. [65]. These techniques include those based on automated test generation with an oracle (Evosuite [66], Randoop [67], and DiffTGen [15]), techniques based on program invariants inferred from the ground truth program (Daikon) [88], an automated test generation with crash and memory safety oracle (Opad) [13], techniques based on the similarity between the original and patched programs (Patch-sim [25], S3 [22], ssFix [24], and CapGen [23]), and techniques based on anti-patterns [18].

4) RQ5: Evaluation concerning APR patches generated with perfect fault localization

We evaluate the performance of MIPI on the APR patches that are generated with perfect fault localization in our dataset. We considered a patch to be a perfect fault localization patch if it modifies only the buggy methods (i.e., the methods modified by the developer to fix the bug). Table 4 shows the detailed numbers of perfect fault localization patches in our dataset.

Forcing a test oracle (e.g., human or a correct program version) is impractical in most real-world settings. We thus focus only on comparing oracle-free techniques such as Anti-patterns [18], Patch-Sim [25], and Opad [13]. We obtained their results from the study of Wang et al. [65] (i.e., use the Defects4J patches), as similar to RQ4 but only focusing on the patches generated with perfect fault localization. For a fair comparison, we also evaluate MIPI on the same set of patches that were used to evaluate the existing techniques.

We further investigate the distribution of patches filtered by MIPI and the existing techniques to see how MIPI is complementary to these techniques.

TABLE 4: Patches generated with perfect fault localization

| Benchmark   | Correct | Incorrect | Total |
|-------------|---------|-----------|-------|
| QuixBugs    | 158     | 158       | 277   |
| Defects4J   | 236     | 236       | 587   |
| Overall     | 394     | 394       | 864   |

FIGURE 4: Comparing the distribution of the meaning distance from the method name to the buggy and correct method body codes. WSR test is significant, p-value = 0.002 and effect size = 0.087

A. RESULTS OF RQ1: IS THE MEANING OF THE METHOD NAME MORE SIMILAR TO THE MEANING OF CORRECT METHOD BODIES THAN TO THE MEANING OF BUGGY METHOD BODIES?

Fig. 4 shows the distribution of the meaning distances from the method name to the buggy method body (blue) and to the correct method body (orange). As shown in Fig. 4, the median value of the distance from method names to buggy method bodies (i.e., 0.014) is larger than that of the correct method bodies (0.010).

To test if the two median values are significantly different, we used the Wilcoxon signed-rank (WSR) test [89] to test the null hypothesis that "the meaning distance from the method name to the buggy code is less than that to the correct code". The WSR test is a non-parametric statistical hypothesis test for comparing two related samples. The p-value computed by the WSR test is 0.002; hence, the null hypothesis can be rejected at a confidence level of 95%. Supporting the alternative hypothesis that "the median value of the meaning distance from the method name to the buggy method body is greater than the median value of the meaning distance from the method name to the correct method body". Thus, we can statistically claim that "the meaning distance from method name to the buggy method body code is statistically greater than that to the correct method body code".

We computed the Cohen’s d effect size to further investigate the strength of the difference between the two median values. While the WSR test shows how statistically significant two distributions are different, the effect size shows how substantially different they are. The value of Cohen’s d effect size is 0.087, which is very small, indicating that the difference (i.e., the gap) between the meaning distance score of the buggy codes and that of the correct codes is quite small.

Answer for RQ1: The meaning distance from method names to correct method bodies is statistically smaller than to buggy method bodies. However, the gap between the two meaning distances is very small (i.e., the effect size is very small).
Corrections to APR-generate patches are often smaller. This result implies that the APR correctly patched method bodies are statistically different from method names to original method bodies and to APR. We confirmed that the median value of the meaning distances of the two meaning distances. The WSR test confirms that this difference is statistically significant (p-value of <0.001). Cohen’s d effect size value is 0.35, which is small. This result implies that the APR correctly patched method code bodies are often more similar to the method name in comparison to the original method body.

Fig. 5 plots the distribution of the meaning distances from the method name to the original method body (blue) and to the patched method body (orange) of APR correctly patched methods in our dataset. The median value of the distance from the method name to the original method body, i.e. 0.03, is larger than to the patched method body, i.e. 0.027. Similar to the RQ1, we applied the WSR test to compare the distributions of the two meaning distances. The WSR test confirmed that the median value of the meaning distances from method names to original method bodies and to APR correctly patched method bodies are statistically different, with a p-value of <0.001. Cohen’s d effect size value is 0.35, which is small. This result implies that the APR correctly patched method code bodies are often more similar to the method name in comparison to the original method body.

B. RESULTS OF RQ2: TO WHAT EXTENT CAN THE MEANING DISTANCE FROM THE METHOD NAME TO THE ORIGINAL METHOD BODY CLASSIFY THE CORRECTNESS OF APR-GENERATED PATCHES?

Fig. 5 plots the distribution of the meaning distances from the method name to the original method body (blue), and to the patched method body (orange) of APR correctly patched methods in our dataset. The median value of the distance from the method name to the original method body, i.e. 0.03, is larger than to the patched method body, i.e. 0.027. Similar to the RQ1, we applied the WSR test to compare the distributions of the two meaning distances. The WSR test confirmed that the median value of the meaning distances from method names to original method bodies and to APR correctly patched method bodies are statistically different, with a p-value of <0.001. Cohen’s d effect size value is 0.35, which is small. This result implies that the APR correctly patched method code bodies are often more similar to the method name in comparison to the original method body.

Fig. 6 plots the distribution of the meaning distances from the method name to the original method body (blue) and to the patched method body (orange) of all APR incorrectly patched Java methods in our dataset. The median value of the distance from the method name to the original method body (i.e., 0.014) is smaller than to the patched method body (i.e., 0.023). The WSR test confirms that this difference is statistically significant (p-value of <0.001). Cohen’s d effect size value is 0.26, which is small. This result implies that the APR incorrectly patched code bodies are often less similar to the method name in comparison to the original method body.

Answer to RQ2: The meaning distance from the method name to the original method body is often larger than to the APR correctly patched method bodies, but often smaller than to the APR incorrectly patched method bodies. Thus, the meaning distance from the method name to the original method body can be considered an effective threshold for classifying correct and incorrect patches.

C. RQ3: HOW EFFECTIVE IS OUR APPROACH IN IDENTIFYING INCORRECT PATCHES?

Table 5 shows the performance of MIPI in identifying the overfitting patches generated by the APR tools in our dataset. The top section of the table shows the detailed results of MIPI for different projects, whereas the bottom section shows the detailed results concerning different APR tools. The last row summarizes the overall performance of MIPI in the dataset. In the table, the first column shows the names of the projects and APR tools in our dataset. The next four columns show the number of true-positive (TP), false-positive (FP), true-negative (TN), and false-negative (FN) classifications made by MIPI. The last columns show the precision ("Pre."), sensitivity ("Sens."), specificity ("Spec."), and destructive ratio ("DR") scores.

Overall, MIPI filtered 281 (254+27) out of the 1,191 patches in the dataset, of which 254 patches were actually overfitting patches (TP) and 27 were correct patches (FP). It achieves a precision of 90%, a sensitivity of 32%, and a specificity of 93% on the overall dataset. This means that the probability that a patch filtered by MIPI is actually an overfitting patch is approximately 90%. MIPI removed 32% of the overfitting patches from the dataset while preserving 93% of the correct patches generated by APR tools. The destructive ratio score of MIPI on the overall dataset is 0.22, indicating that the percentages of filtered correct patches is approximately one-fifth of the percentages of filtered overfitting patches.

There are 543 overfitting patches that MIPI does not filter out (FN), and 27 correct patches were mistakenly filtered out by MIPI (FP). Among 543 FN cases, 186 overfitting patches were not filtered because the method name that the Code2Vec model predicted for the patched code is exactly the name given by the developer (the patched code is extremely close to the developer intention). An example is a patch generated by TBBar for bug Chart12, shown in Figure 7. Although the patched code is incorrect, the Code2Vec model still predicted exactly the developer given method name for the method, which is “haslistener”. Therefore, MIPI classified
null boolean hasListener(EventListener listener) {
    List list = Arrays.asList(this.listenerList.getListenerList());
    if (list.contains(listener)) {
        return list.contains(listener);
    }
}

FIGURE 7: An overfitting patch generated by TBar.

Among the remaining 357 (i.e., 543-186) FN cases that the patched code is far from the developer’s intention, 59 overfitting patches were not filtered because the predicted method name of the original and the patched code are identical (the original and the patched codes have equal distance to the developer’s intention). This problem occurred when the size of the abstract syntax tree (AST) of the modified method was too large, and Code2Vec did not include the modified code elements in the representation of the method. Specifically, Code2Vec just sampled some part of the source code to feed into the model. However, the sampled data did not contain the modified code element. Therefore, the result for the original and the patched code will be identical. In other words, Code2Vec represented both the original and the patched method bodies by an identical set of AST paths.

The remaining 298 overfitting patches were not filtered because their patched code is determined more similar to the method name (more likely to be correct) in comparison to the original code. There are 27 correct patches that were mistakenly filtered out by MIPI because the patched codes of these patches were determined less similar to the method name (far from developer intention) in comparison to the original code. There are error cases due to the out of vocabulary (OOV) problem of the Code2Vec, that is, the correct method name for a method is out of the vocabulary of the Code2Vec model. Although the size of the vocabulary of the Code2Vec is large (about 300,000 different method names), the OOV problem may still occur. For example, the method name "getlenclosingfunctiondependency/scope" (found in patch Closure114-RSRepair), are not in the vocabulary of Code2Vec and thus will never be predicted. OOV is a common issue in natural language processing. Using subword tokenization techniques, i.e. Byte-Pair Encoding [90], is a possible way to alleviate this issue [91], [92].
TABLE 6: Comparing the results of MIPI with existing techniques on the QuixBugs dataset. (* denotes the technique requires test oracles or the correct program)

| Method        | TP  | FP  | TN  | FN  | Pre. | Sens. | Spec. | DR   |
|---------------|-----|-----|-----|-----|------|-------|-------|------|
| Evosuite*     | 171 | 0   | 158 | 6   | 1.00 | 0.97  | 1.00  | 0.00 |
| InputSampling*| 112 | 0   | 158 | 65  | 1.00 | 0.63  | 1.00  | 0.00 |
| Daikon*       | 164 | 125 | 33  | 13  | 0.57 | 0.93  | 0.21  | 0.85 |
| MIPI          | 114 | 4   | 154 | 63  | 0.97 | 0.64  | 0.97  | 0.04 |

TABLE 7: Comparing the results of MIPI with existing techniques on the Defects4J dataset. (* denotes the technique requires test oracles or the correct program)

| Method | TP  | FP  | TN  | FN  | Pre. | Sens. | Spec. | DR   |
|--------|-----|-----|-----|-----|------|-------|-------|------|
| MIPI   | 140 | 23  | 213 | 480 | 0.86 | 0.23  | 0.90  | 0.43 |
| S3     | 516 | 135 | 113 | 138 | 0.79 | 0.79  | 0.46  | 0.69 |
| ssFix  | 515 | 138 | 110 | 139 | 0.79 | 0.79  | 0.44  | 0.71 |
| CapGen | 506 | 140 | 108 | 148 | 0.78 | 0.78  | 0.44  | 0.73 |
| Patch-Sim | 249  | 51  | 186 | 392 | 0.83 | 0.39  | 0.78  | 0.55 |
| E-Patch-Sim | 166  | 36  | 202 | 457 | 0.82 | 0.26  | 0.85  | 0.59 |
| Anti-patterns | 219 | 37 | 211 | 435 | 0.86 | 0.33  | 0.85  | 0.45 |
| Daikon* | 337 | 38 | 166 | 120 | 0.90 | 0.74  | 0.81  | 0.25 |

FIGURE 8: Distribution of patches in the QuixBugs dataset filtered by different techniques

high. As shown in Table 6, Evosuite and InputSampling can identify overfitting patches with 100% precision, and do not mistakenly filter out any correct patches in the QuixBugs dataset. MIPI is comparable to InputSampling in terms of sensitivity; however, it is worse than InputSampling in terms of precision and specificity. Note that, for identifying the patch correctness, Evosuite, InputSampling, and Daikon require a correct program version, while MIPI does not.

We further investigated how MIPI can complement existing techniques. Figure 8 shows the distribution of patches filtered by the MIPI and the existing techniques. There are 66 overfitting patches that are commonly filtered by the MIPI and all considered techniques. The number of overfitting patches that MIPI filtered but InputSampling, Evosuite, and Daikon cannot are 42, 1, and 8, respectively.

2) Comparison results on the Defects4J dataset

We compared the results of MIPI on the Defects4J dataset with the results of existing techniques reported in the study by Wang et al. [65]. Table 7 presents the comparison results.

As shown in the table, the proposed approach, MIPI, correctly filters out 140 (23% of 620) overfitting patches while wrongly discards 23 (10% of 236) correct patches in the Defects4J dataset. It achieved a precision of 86%.

Compared to similarity-based techniques, MIPI has a lower sensitivity score but outperforms all these similarity-based techniques in terms of precision, specificity, and destructive ratio. MIPI achieves a higher precision score (i.e., 86%) than all similarity-based techniques (i.e., from 79% to 83%), which means a patch filtered by MIPI has a higher probability of being an overfitting patch than that of a patch filtered by the similarity-based techniques. Both syntactic (S3, ssFix, and CapGen) and behavior (Patch-sim and E-patch-sim) similarity-based techniques can filter out more overfitting patches (i.e., higher sensitivity); however, they wrongly filter out many more correct patches (i.e., lower specificity) than MIPI. Specifically, the syntactic-similarity-based techniques filtered out from 77% to 79% of the overfitting patches in the dataset, which is three times larger than the 23% of MIPI. However, they wrongly filtered from 54% to 56% of correct patches, which is five times larger than MIPI, because MIPI mistakenly filtered only 10% of the correct patches. Behavior-similarity-based techniques can filter out from 26% (E-patch-sim) to 39% (Patch-sim) of the overfitting patches in the dataset, which is 1.1 to 1.7 times larger than the 23% of MIPI. However, they wrongly filtered from 15% to 22% of correct patches, which is 1.5
to 2.2 times more than the 10% of MIPI. The destructive ratio of MIPI (i.e., 0.43) is significantly lower than that of both syntactic-similarity-based (i.e., from 0.69 to 0.73) and behavior-similarity-based techniques (i.e., from 0.55 to 0.59), which indicates that MIPI cause less destructive effects than similarity-based techniques when they are applied to filter overfitting patches.

Compared to Anti-patterns, a technique based on a set of predefined syntactic patterns. Both MIPI and Anti-patterns achieved a precision of 86%. Anti-patterns achieves a higher specificity than MIPI (33% versus 23%), which means that anti-patterns can filter out more overfitting patches than MIPI. However, MIPI has higher specificity than Anti-patterns (90% versus 85%), which means that MIPI can preserve more correct patches. The destructive ratio of MIPI (i.e., 0.43) is smaller than that of Anti-patterns (i.e., 0.45), indicating that MIPI causes less destructive effects than anti-patterns when they are applied to filter overfitting patches.

Compared to dynamic invariants-based technique, Daikon. MIPI has higher specificity than Daikon (i.e., 90% versus 81%), indicating that MIPI retained more correct patches than Daikon. Daikon outperforms MIPI in terms of precision (i.e., 90% versus 86%), sensitivity (i.e., 74% versus 23%), and destructive ratio (i.e., 0.25 versus 0.43). This indicates that Daikon can filter out more overfitting patches with higher precision and less destructive effects than MIPI. However, to infer the set of dynamic program invariants, Daikon requires the availability of the correct version which is often not available in practical settings.

Compared to patch assessment techniques that are based on automated test generation. Because the patch assessment techniques based on automated test generation (e.g., Evo-suite, Randoop, DiffTGen, and OPad) are deterministic, it is obvious that they can achieve extremely high precision and specificity, and their destructive ratio value is close to zero. Our approach, MIPI, and other non-deterministic techniques are incomparable to these automated testing-based techniques in terms of reliability and safety when filtering patches. In terms of sensitivity, MIPI outperforms E-Opad and R-Opad (which are techniques that do not require oracle information), but lower than EvoSuite, Randoop, and DiffTGen (which are techniques that require oracle information to determine test results).

We further investigated the distribution of patches filtered by MIPI and existing techniques to see how MIPI is complementary to these techniques. Because requiring the availability of a test oracle (e.g., human or a corrected program version) is impractical in most real-world settings, we focus mainly on the comparison of the oracle-free techniques. Figure 8 shows the distribution of the patches filtered by MIPI and existing oracle-free techniques on the Defects4J dataset. As shown in the figure, only three overfitting patches are commonly identified by all displayed techniques, whereas many overfitting patches are exclusively detected by different techniques. There are 158 overfitting patches that cannot be filtered by any oracle-free techniques. MIPI exclusively filtered out 45 overfitting patches, and also exclusively filtered 16 correct patches.

Table 8 presents the details of the number of overfitting patches that MIPI commonly and exclusively detected with respect to each existing technique. Considering the first row, for example, there are 41 overfitting patches detected by MIPI that Anti-patterns also detected, and 99 overfitting patches that MIPI detected but Anti-patterns did not. As shown in the table, MIPI can filter out 52 to 132 additional overfitting patches that each existing technique cannot filter out. The results indicate that MIPI is highly complementary to the existing techniques.

### E. RESULTS OF RQ5: EVALUATION ON PATCHES WITH PERFECT FAULT LOCALIZATION

Table 9 shows the performance of MIPI on APR patches generated with perfect fault localization in QuixBugs, Defects4J, and the overall dataset. Considering the QuixBugs benchmark, for example, MIPI correctly filters 74 out of 119 overfitting patches (sensitivity of 62%) but mistakenly filters 4 out of 158 correct patches (specificity of 97%). It achieves a precision of 95% on the QuixBugs benchmark. Overall, MIPI filtered 145 (31%) out of a total of 470 overfitting patches with a precision of 84% while preserving 93% (367/394) of the correct patches.
TABLE 9: Performance of MIPI on perfect fault localization patches

| Benchmark | TP | FP | TN | FN | Pre. | Sens. | Spec. | DR |
|-----------|----|----|----|----|------|-------|-------|----|
| QuixBugs  | 74 | 4  | 154| 45 | 0.95 | 0.92  | 0.97  | 0.05|
| Defects4J | 71 | 23 | 213| 280| 0.76 | 0.20  | 0.90  | 0.48|
| Overall   | 145| 27 | 367| 325| 0.84 | 0.31  | 0.93  | 0.25|

TABLE 10: Comparison of oracle-free techniques on perfect fault localization patches

| Method    | TP  | FP  | TN  | FN  | Pre. | Sens. | Spec. | DR  |
|-----------|-----|-----|-----|-----|------|-------|-------|-----|
| MIPI      | 71  | 23  | 213 | 280 | 0.76 | 0.20  | 0.90  | 0.48|
| Anti-patterns | 108 | 37  | 199 | 243 | 0.74 | 0.31  | 0.84  | 0.51|
| Patch-Sim | 129 | 51  | 185 | 222 | 0.72 | 0.37  | 0.78  | 0.59|
| E-Patch-Sim | 76  | 36  | 200 | 275 | 0.68 | 0.22  | 0.85  | 0.70|
| R-Opad    | 20  | 0   | 236 | 331 | 1.00 | 0.06  | 1.00  | 0.00|
| E-Opad    | 45  | 0   | 236 | 306 | 1.00 | 0.13  | 1.00  | 0.00|

Table 10 compares the results of MIPI with existing oracle-free techniques on APR patches with perfect fault localization. The experimental results show that MIPI achieves higher precision and preserves more correct patches than heuristic-based techniques (Anti-patterns, Patch-Sim, E-Patch-Sim) while filtering more overfitting patches than the automated test generation based techniques (OPad, E-OPad) which do not have access to the test. Specifically, MIPI achieves the precision of 76% and specificity of 90%, which is the highest among the heuristic-based techniques including Anti-patterns, Patch-Sim, and E-patch-sim. The number of overfitting patches that MIPI filtered is 71 while that of R-Opad and E-Opad is 20 and 45, respectively.

Table 11 shows the number of overfitting patches that MIPI commonly and exclusively filtered for each existing technique. Considering the first row, for example, there are 24 overfitting patches that both MIPI and Anti-pattern filtered, and 47 overfitting patches that MIPI filtered but Anti-pattern did not. The table shows that MIPI can filter out from 44 to 69 additional overfitting patches that each existing oracle-free technique cannot filter out. The results indicate that MIPI is highly complementary to existing APCA techniques.

VII. DISCUSSIONS

A. CAPABILITY OF FILTERING BLOCKING OVERFITTING PATCHES

An APR run (a running of an APR tool with a buggy program) is usually configured to stop after generating at most K plausible patches or timeout. After the APR tool stops by the first constraint, all K patches may be overfitting patches. In this case, the APR run is said to be blocked by the overfitting patches (i.e., the overfitting patches have blocked the APR tool from generating the correct patch). Filtering the blocking overfitting patches to unblock the APR run is a prerequisite for the APR to generate the correct patch. Automated patch correctness assessment (APCA) techniques can unblock the blocked APR run by filtering the blocking overfitting patches.

Our dataset contains a total of 738 APR runs, in which 540 APR runs generate overfitting patches. Among them, 508 APR runs generate only overfitting patches, and 32 APR runs generate both overfitting and correct patches.

The 508 APR runs that generate only overfitting patches are considered blocked APR runs, as the overfitting patches have blocked them from generating the correct patch (i.e., it makes the APR tool stop before a correct patch can be generated). MIPI has filtered at least one blocking overfitting patch for 23% (117/508) of the blocked runs, in which there are 110 cases (22% of the blocked runs) that MIPI filtered all the generated overfitting patches (i.e., MIPI unblocked for 22% of the blocked runs in the dataset).

In 32 APR runs that generate both overfitting and correct patches in our dataset, MIPI filtered at least one patch for 9 APR runs. Among these, there are 5 cases that MIPI filtered only overfitting patches, no case that MIPI filtered only correct patches, and 4 cases that MIPI filtered both overfitting and correct patches. We cannot say for sure whether MIPI has improved the ranking of the correct patches because the information about the original ranking of the patches is not available. However, it is likely that the ranking of the correct patch is improved or at least unchanged. Specifically, in the 5 cases that MIPI filtered only overfitting patches, the ranking of the correct patch will be improved, except the case that all the filtered overfitting patches are ranked by the APR tool lower than the correct patch (in this case, the ranking of the correct patch will be unchanged). There is no case that MIPI filtered only correct patches, thus, it is less likely that the MIPI will worsen the ranking of the correct patch.

Compared to existing oracle-free APCA techniques, MIPI can unblock for more blocked APR runs than automated test generation based techniques (R-Opad, and R-Opad) and causes damage to fewer APR runs than the heuristic-based techniques. In addition, MIPI did not cause any APR runs in the dataset to become blocked while both Anti-patterns, Patch-Sim, and E-Patch-Sim did for one APR run. More detailed results and discussions are provided in Appendix.

B. THREATS TO VALIDITY

1) Threat to external validity

One threat to the external validity is whether our approach can be generalized to different types of programs and APR tools. Because QuixBugs contains a set of small-sized programs implementing classical algorithms, whereas Defects4J contains large real-world programs solving industrial problems, we believe that these programs can cover a wide range...
of program types. The patches in our dataset are generated by a large number of existing APR tools covering all main categories of APR, including heuristic-search-, semantic-search-, and machine-learning-based APRs.

Another threat to the external validity is that our proposed approach can be varied depending on the quality of method names given by the developers. As developers of different qualifications may vary in their naming skills, the developers may use method names that do not reflect the intended behavior of the code. However, it is common for software projects and companies to have their own coding conventions for naming program entities [34], [35]. Also, some studies [35]–[37] have mentioned and studied that the method name usually contains important hints about the intended behaviors of the method. Therefore, we believe the proposed method is generally good for general software projects. Also, the experimental results confirmed that MIPI is as good as or better than other approaches in both large real-world projects, Defects4J, and small programs in education, QuixBugs.

2) Threat to internal validity
As a threat to the validity, we have discarded some patches from our dataset, either because they are the patches of deprecated bugs, or are patches that are not supported by tools used in our implementation (e.g., Code2Vec cannot parse the constructor methods of Java classes). However, the number of removed patches was small compared to the whole dataset. We made all of our experimental data and implementation available online at https://bit.ly/mipiexp.

3) Threat to construct validity
There are a variety of code understanding models and word embedding models. Thus, there are many different choices for our implementation. For example, we can implement the Code Description Generator component based on another code understanding model, instead of Code2Vec. We can also implement the Meaning Similarity Measurer component using another word embedding model, instead of Bert, and using a different similarity metric instead of the cosine distance. In addition, we can use a more sophisticated approach to extract the description of the developer’s intention instead of using only the method name as our current implementation. However, the current study does not explore all possible design decisions and leaves them for future work.

VIII. CONCLUSION
We proposed MIPI, a novel approach to identify incorrect patches in APRs based on the semantic meaning of the source code. The valuable finding of our study is that the developer intention extracted from the method name is effective to distinguish the incorrect patch. MIPI is more precise and less destructive than existing heuristic-based patch assessment techniques, whereas it can filter out more overfitting patches than automated test generation techniques that do not require oracle information. In addition, MIPI is highly complementary to existing automatic patch-correctness assessment techniques.

APPENDIX. COMPARING TO EXISTING ORACLE-FREE APCA TECHNIQUES CONCERNING BLOCKING OVERFITTING PATCHES
Table 12 compares the ability to filter blocking overfitting patches of MIPI and existing oracle-free APCA techniques. We obtained the results of the existing techniques from the study of Wang et.al. [65] (i.e., the Defects4J benchmark).

In the dataset, 493 runs are blocked APR runs that generate only overfitting patches (see the column "Total" of the section "Blocked APR Runs"). The column "Unblocked" shows the number of blocked APR runs that each APCA technique can unblock by filtering out all the generated overfitting patches. A higher Unblocked number means increasing the likelihood of creating more correct patches. MIPI can unblock more number of blocked APR runs than automated testing based techniques (i.e., 107 versus 40 and 60 of R-Opad and E-Opad, respectively) but less than heuristic-based techniques (i.e., 107 versus 179, 190, and 133 of Anti-patterns, Patch-Sim, and E-Patch-Sim, respectively).

Section "Successful APR Runs" in Table 12 shows the results concerning APR runs that all of the generated patches are correct. Totally, 183 APR runs generated only correct patches. The column "Damaged" shows the number of APR runs that the APCA techniques filtered all the generated correct patches. It is acceptable that the APCA technique filters out some but not all of the correct patches generated by an APR run. However, it is severely damaged if the APCA technique filters all the correct patches that the APR tool generated. It is because we are unsure if the APR tool can generate another correct patch. The results showed that MIPI caused damage to fewer APR runs than heuristic-based techniques (i.e., 17 versus 23, 33, and 28 of Anti-patterns, Patch-Sim, and E-Patch-Sim, respectively). The automated test generation-based techniques (R-Opad, and E-Opad) do not cause damage to any APR runs.

Section "APR Runs with both Overfitting and Correct Patches" in Table 12 shows the results on the APR runs that generated both overfitting and correct patches. 31 APR runs have generated both overfitting and correct patches. Columns "Filter OP", "Filter CP" and "Filter Both" show the number of runs that the APCA techniques filtered only overfitting patches, filtered only correct patches, and filtered both overfitting and correct patches, respectively. Column "No CP Remains" shows the number of APR runs that the APCA technique filtered all the generated correct patches. Column "Blocked" shows the number of APR runs that the APCA techniques filtered all correct patches but at least one overfitting patch remains (i.e., the APR run becomes blocked). This case is the most serious damage case because the remaining overfitting patch will block the APR runs from generating another correct patch.

Since the information about the original ranking of the patches is not available, we are not sure if an overfitting patch is generally good for general software projects. Also, the experimental results confirmed that MIPI is as good as or better than other approaches in both large real-world projects, Defects4J, and small programs in education, QuixBugs.
TABLE 12: Comparing the capability of filtering blocking overfitting patches of oracle-free techniques

| Method   | Blocked APR Runs Total | Unblocked | Successful APR Runs Total | Damaged | APR Runs with both Overfitting (OP) and Correct (CP) Patches Total | Filter OP | Filter CP | Filter Both | No CP Remains Blocked |
|----------|------------------------|-----------|---------------------------|---------|------------------------------------------------------------------|---------|---------|---------------|------------------------|
| MIPI     | 493                    | 107       | 183                       | 17      |                                                                  | 3       | 0       | 3            | 0                      |
| Anti-pattern | 493                    | 179       | 183                       | 23      |                                                                  | 4       | 1       | 4            | 1                      |
| Patch-Sim | 493                    | 190       | 183                       | 33      |                                                                  | 9       | 0       | 9            | 1                      |
| E-Patch-Sim | 493                    | 133       | 183                       | 28      |                                                                  | 6       | 0       | 6            | 1                      |
| R-Opad   | 493                    | 40        | 183                       | 0       |                                                                  | 0       | 0       | 0            | 0                      |
| E-Opad   | 493                    | 69        | 183                       | 0       |                                                                  | 0       | 0       | 0            | 0                      |

is generated before or after the correct patch. However, the ranking of the correct patch is likely to be improved if the APCA technique filters only overfitting patches, except the cases that all the filtered overfitting patches are generated after the correct patch (in this case, the ranking of the correct patch will be unchanged). The results showed that MIPI is likely to improve the ranking of the correct patches for 5 APR runs. MIPI is less than Anti-Patterns and Patch-Sim but better than R-Opad. The ranking of the correct patch is likely to worsen if the APCA technique filters only correct patches. There is no case that MIPI filtered only correct patches thus, it is less likely that the MIPI will worsen the ranking of the correct patch. As column "No CP Remains" showed, MIPI less often causes damage to the APR runs (i.e., filtered all the generated correct patches) than all the heuristic-based techniques. In addition, as shown in the column "Blocked", MIPI does not cause any APR runs to become blocked while both Anti-patterns, Patch-Sim, and E-Patch-Sim make one APR run become blocked.

REFERENCES

[1] M. Lake, “Epic failures: 11 infamous software bugs,” sep 2010.
[2] M. Harris, “Google reports self-driving car mistakes: 272 failures and 13 near misses;,” 2016.
[3] M. Martinez and M. Monperrus, “Astar: Exploring the design space of generate-and-validate program repair beyond GenProg,” Journal of Systems and Software, vol. 151, pp. 65–80, may 2019.
[4] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, “SupFix: Automated End-To-End Repair at Scale,” Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2019, pp. 269–278, 2019.
[5] D. Campos, A. Restivo, H. Sereno Ferreira, and A. Ramos, “Automatic Program Repair as Semantic Suggestions: An Empirical Study,” in 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), no. 2. IEEE, apr 2021, pp. 217–228.
[6] M. Motwani, M. Soto, Y. Brun, R. Just, and C. Le Goues, “Quality of Automated Program Repair on Real-World Defects,” IEEE Transactions on Software Engineering, vol. 5589, no. c, pp. 1–1, 2020.
[7] T. Durieux, Y. Hamadi, M. Monperrus, T. Durieux, Y. Hamadi, M. Monperrus, P.-d. P. Generation, and T. Durieux, “Production-Driven Patch Generation,” in ICSE, no. May, 2017.
[8] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, “Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset,” Empirical Software Engineering, pp. 1–29, 2016.
[9] Y. Tao, J. Kim, S. Kim, and C. Xu, “Automatically generated patches as debugging aids: a human study,” in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014. New York, New York, USA: ACM Press, 2014, pp. 64–74.
[10] 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 - ESEC/FSE 2015. New York, New York, USA: ACM Press, 2015, pp. 532–543.
[11] J. P. Cambromero, J. Shen, J. Cito, E. Glassman, and M. Rinard, “Characterizing developer use of automatically generated patches,” pp. 181–185, 2019.
[12] Z. Yu, M. Martinez, B. danglot, T. Durieux, and M. Monperrus, “Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system,” Empirical Software Engineering, may 2018.
[13] J. Yang, A. Zhikharsiev, Y. Liu, and L. Tan, “Better test cases for better automated program repair,” in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, no. 1. ACM, aug 2017, pp. 831–841.
[14] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, “Identifying Patch Correctness in Test-Based Automatic Program Repair,” ICSE, 2018.
[15] Q. Xin and S. P. Reiss, “Identifying test-suite-overfitted patches through test case generation,” Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2017, pp. 226–236, 2017.
[16] Z. Yu, M. Martinez, B. Daglot, T. Durieux, and M. Monperrus, “Test Case Generation for Program Repair: A Study of Feasibility and Effectiveness,” Empirical Software Engineering, 2018.
[17] F. Long and M. Rinard, “An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems,” Proceedings of the 38th International Conference on Software Engineering, pp. 702–713, 2016.
[18] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, “Anti-patterns in search-based program repair,” Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016, pp. 727–738, 2016.
[19] B. Yang and J. Yang, “Exploring the Differences between Plausible and Correct Patches at Fine-Grained Level,” in 2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF). IEEE, feb 2020, pp. 1–8.
[20] 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 - POPL 2016, New York, New York, USA: ACM Press, 2016, pp. 298–312.
[21] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, “Shaping program repair space with existing patches and similar code,” in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2018, pp. 298–309.
[22] X. B. D. Le, D. H. Chu, D. Lo, C. Le Goues, and W. Visser, “S3: Syntax-and semantic-guided repair synthesis via programing by examples,” in Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, vol. Part F1301, 2017, pp. 593–604.
[23] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Chen, “Context-Aware Patch Generation for Better Automated Program Repair,” Icse, vol. 18, 2018.
[24] X. Xiong and S. P. Reiss, “Leveraging syntax-related code for automated program repair,” in ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, 2017.
[25] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, “Identifying Patch Correctness in Test-Based Automatic Program Repair,” ICSE, 2018.
[26] A. Ghanbari, “ObjSim: Lightweight Automatic Patch Prioritization via Object Similarity,” in ISSTA, 2020.
[27] H. Tian, L. Li, K. Liu, J. Klein, A. K. Kaboré, A. Koyuncu, and T. F. Bisalyné, “Evaluating representation learning of code changes for predicting patch correctness in program repair,” Tech. Rep., 2020.
[28] V. Cuvuık, D. Horvath, F. Horvath, and L. Vidacs, “Utilizing Source Code Embeddings to Identify Correct Patches,” in 2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF). IEEE, feb 2020, pp. 18–25.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2022.3145983, IEEE Access
[29] B. Lin, S. Wang, M. Wen, and X. Mao, “Context-Aware Code Change Embedding for Better Patch Correctness Assessment,” Journal of the ACM, vol. 1, pp. 54–72, 2021.

[30] 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, jan 2021.

[31] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao, “How Different Is It Between Machine-Generated and Developer-Provided Patches? An Empirical Study,” in The Correct Patches Generated by Automated Program Repair Techniques, 2016.

[32] V. Csurvik, D. Horváth, M. Lajkó, and L. Vídac, “Exploring Plausible Patches Using Source Code Embeddings in JavaScript,” in 2nd International Workshop on Automated Program Repair, 2021.

[33] 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 Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019. Institute of Electrical and Electronics Engineers Inc., apr 2019, pp. 102–113.

[34] R. C. Martin, Clean Code. Prentice Hall, 2008.

[35] M. Fowler, K. Beck, J. Braat, W. Opdyke, D. Roberts, and E. Gamma, Refactoring: Improving the Design of Existing Code, 1st ed. Addison-Wesley Professional, 1999.

[36] R. S. Alshuwaikhat, C. D. Newman, M. J. Decker, M. L. Collard, and J. T. Maletic, “On the Naming of Methods: A Survey of Professional Developers,” in ICSE 2021, 2021.

[37] B. Lilbit, A. Bégel, and E. Swerser, “Cognitive Perspectives on the Role of Naming in Computer Programs,” Proceedings of the 18th Annual Psychology of Programming Interest Group Workshop, no. September, pp. 53–67, 2006.

[38] T. Ben-nun, A. S. Jakobovits, and T. Hoefle, “Neural Code Comprehension: A Learnable Representation of Code Semantics,” in 32nd Conference on Neural Information Processing Systems, no. NeurIPS, 2018.

[39] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “code2vec: Learning Distributed Representations of Code,” in POPL 2019, vol. 3, no. January, 2019.

[40] E. Yahav, S. Brody, O. Levy, and E. Yahav, “code2seq: Generating sequences from structured representations of code,” in ICLR 2019, no. 1, 2019, pp. 1–19.

[41] J. Henkel, S. K. Lahiri, B. Lilbit, and T. Reps, “Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces,” in ESEC/FSE, 2018.

[42] E. Yahav, “From Programs to Interpretable Deep Models and Back,” in International Conference on Computer Aided Verification. Springer International Publishing, 2018, pp. 27–37.

[43] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in Proceed- ings of the 2019 Conference of the North, vol. 1. Stroudsburg, PA, USA: Association for Computational Linguistics, 2019, pp. 4171–4186.

[44] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé, “LSRepair : Live Search of Fix Ingredients for Automated Program Repair,” in 25th Asia-Pacific Software Engineering Conference, 2018.

[45] M. Monperrus and Martin, “Automatic Software Repair,” ACM Computing Surveys, vol. 51, no. 1, pp. 1–24, 2018.

[46] Y. Yuan and W. Banzhaf, “ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming,” 2018.

[47] S. Mehthaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, “Semantic Program Repair Using a Reference Implementation,” in ICSE, 2019.

[48] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, “ELIXIR : Effective Object-Oriented Program Repair,” in ASE 2017, 2017, pp. 648–659.

[49] L. Chen, Y. Pei, and C. A. Furia, “Contract-based program repair without the contracts,” ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp. 673–674, 2017.

[50] J. Xu, M. Martínez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, “Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs,” IEEE Transactions on Software Engineering, vol. PP, no. 99, p. 1, 2016.

[51] X. B. D. Le, D. Lo, and C. L. Goues, “History Driven Program Repair,” 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 213–224, 2016.
