Automated Patch Correctness Assessment: How Far are We?

Shangwen Wang*
wangshangwen13@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Ming Wen*†
mwenaa@hust.edu.cn
School of Cyber Science and Engineering, Huazhong University of Science and Technology
Wuhan, China, ✉

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

Hongjun Wu
wuhongjun15@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Yihao Qin
qinyihao15@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Deqing Zou†‡
deqingzou@hust.edu.cn
School of Cyber Science and Engineering, Huazhong University of Science and Technology
Wuhan, China

Xiaoguang Mao
xgmao@nudt.edu.cn
College of Computer Science, National University of Defense Technology
Changsha, China

Hai Jin†§
hjin@hust.edu.cn
School of Computer Science and Technology, Huazhong University of Science and Technology
Wuhan, China

ABSTRACT

Test-based automated program repair (APR) has attracted huge attention from both industry and academia. Despite the significant progress made in recent studies, the overfitting problem (i.e., the generated patch is plausible but overfitting) is still a major and long-standing challenge. Therefore, plenty of techniques have been proposed to assess the correctness of patches either in the patch generation phase or in the evaluation of APR techniques. However, the effectiveness of existing techniques has not been systematically compared and little is known to their advantages and disadvantages. To fill this gap, we performed a large-scale empirical study in this paper. Specifically, we systematically investigated the effectiveness of existing automated patch correctness assessment techniques, including both static and dynamic ones, based on 902 patches automatically generated by 21 APR tools from 4 different categories. Our empirical study revealed the following major findings: (1) static code features with respect to patch syntax and semantics are generally effective in differentiating overfitting patches over correct ones; (2) dynamic techniques can generally achieve high precision while heuristics based on static code features are more effective towards recall; (3) existing techniques are more effective towards certain projects and types of APR techniques while less effective to the others; (4) existing techniques are highly complementary to each other. For instance, a single technique can only detect at most 53.5% of the overfitting patches while 93.3% of them can be detected by at least one technique when the oracle information is available. Based on our findings, we designed an integration strategy to first integrate static code features via learning, and then combine with others by the *majority voting* strategy. Our experiments show that the strategy can enhance the performance of existing patch correctness assessment techniques significantly.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; Software testing and debugging.

KEYWORDS

Patch correctness, Program repair, Empirical assessment.

ACM Reference Format:
Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated Patch Correctness Assessment: How Far are We?. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416590

*The first two authors contributed equally to this work, and Ming Wen is the corresponding author.
†National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, HUST, Wuhan, 430074, China
‡Shenzhen HUST Research Institute, Shenzhen, 518057, China
§Cluster and Grid Computing Lab, HUST, Wuhan, 430074, China

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416590

1 INTRODUCTION

Automated Program Repair (APR) has gained huge attention from both industry and academia recently. Over the years, substantial APR tools have been proposed [27, 38, 44, 46, 47, 74, 77, 80, 82] with the aim to reduce the excessively high cost in bug fixing. APR tools have shown to be promising towards both practical significance and research value. For instance, SapFix, which was proposed by Facebook to automatically generate and suggest fixes [52], has already been deployed in real products that collectively consist of millions of lines of code and are used by millions of users worldwide.

Despite the tremendous effectiveness achieved, existing APR tools still face a significant and long-standing challenge: *the overfitting problem* [37, 68, 71]. The overfitting problem arises when the measurements used to assess the correctness of an automated generated patch is imperfect. Due to the absence of formal specifications of the desired behavior, most of the APR tools leverage the developer-provided test suite as partial specifications to assess whether a patch is correct currently. Such a measurement assumes a patch that passes all the test cases to be correct, and incorrect otherwise. However, test suites in real world projects are often weak and inadequate [26, 64], and thus a patched program passing all the tests might be simply overfitting to the test suite and still be faulty. Later on, people denote a patch that passes the test suite as a *plausible* patch [64]. A plausible patch that indeed fixes the target bug is deemed *correct*, otherwise is regarded as an *overfitting* patch. As revealed by recent studies [26, 37, 64], existing APR techniques are widely suffering from the overfitting problem, that is they generate more overfitting patches than correct ones on real bugs, thus leading to a *low precision* of their generated patches.

The low precision of the generated patches significantly affected the practical usefulness of existing APR techniques. Therefore, growing research efforts have been made to identify correct patches among plausible ones automatically [33, 34, 70, 77, 79, 81, 85, 86, 88]. For instance, `DiffTGen` was initially proposed to identify overfitting patches through automated test generation based on the developer-provided patch [79], which is denoted as the **oracle patch** in this study. It first tries to enhance the adequacy of the provided test suite via generating an extra set of test cases, and then assumes a generated patch whose test outcome based on such tests differs from that of the oracle patch to be an overfitting patch. To ease our presentation, we denote those techniques designed for Automated Patch Correctness Assessment as **APCA** techniques in this study.

Existing APCA techniques differentiate themselves in diverse design spaces. First, they can be either *static* or *dynamic* depending on whether they execute test cases. Static techniques prioritize or filter out incorrect plausible patches via analyzing the characteristics of patches statically (e.g., `Anti-patterns` [70]). On the contrary, dynamic techniques generally leverage *automated test generation* tools, such as Randoop and Evosuite, to identify correct patches among plausible ones (e.g., `DiffTGen` [79], `PATCH-SIM` [81]). Second, they can be differentiated in whether the oracle patch is required. Those techniques require the oracle patch, such as `DiffTGen` [79], run automated test generation tools based on the **oracle program** (i.e., the program after applying the oracle patch), and then leverage those automated generated tests to identify overfitting patches. Those techniques do not require the oracle patch (e.g., `PATCH-SIM` [81] and `Opad` [86]) generate extra tests based on the buggy version, and then leverage other information or certain heuristics as the oracle to identify overfitting patches. Techniques designed without the oracle patch can be applied to the process of **patch generation** with the aim to prioritize and filter out overfitting patches, and thus to enhance the precision of APR techniques [77, 81]. On the contrary, techniques requiring the oracle patch are often used for **patch evaluation**, that is to assess the effectiveness of APR techniques [33, 88]. Both of these two directions of APCA techniques are reported to be significant to the APR community [33, 79, 81, 86, 88]. For instance, manually annotating the correctness of patches is subjective and rather expensive as reported by existing studies [33, 88], and thus APCA techniques, with the oracle information, are useful to facilitate the evaluation of APR techniques.

Although huge efforts have been made towards automated patch correctness assessment, the effectiveness of existing techniques has not been systematically studied and compared. Besides, little is known to their advantages and disadvantages. A recent study reported that `DiffTGen` and `Randoop` can only identify fewer than a fifth of the overfitting patches through an empirical study based on 189 patches [33]. However, the behind reasons and the effectiveness of other advanced APCA techniques remain unknown. Therefore, there is an urge need for a comprehensive empirical study comparing and analyzing the effectiveness of all the state-of-the-art APCA techniques based on a larger number of patches. Such a study is necessary and essential, which can help us find the answers to important questions when designing APCA techniques. For instance, whether existing APCA techniques are more effective towards certain types of patches or APR techniques? Besides, are existing techniques complementary to each other and whether the integration of them can enhance the performance? Answering such questions can guide researchers to design more effective techniques.

This study aims to bridge this gap, which performs a systematic empirical study for automated patch correctness assessment including 9 different techniques and 3 heuristics based on 8 static code features on the most comprehensive patch benchmark so far (i.e., 902 patches in total). Via investigating how many overfitting patches can be identified by each APCA technique, we understood the effectiveness of existing techniques, including the advantages and disadvantages, and pointed out how they can be improved. For instance, `PATCH-SIM` can be enhanced by test case purification while `Daikon` can be improved by considering the characteristics of invariants. Our study also makes the following important findings:

- F1: Heuristics based on static features gauging patch syntax and semantics are generally effective in differentiating overfitting patches over correct ones, which can achieve high recalls.
- F2: Dynamic APCA techniques can achieve high precision while most of them will label correct patches as overfitting. Besides, those techniques with the oracle information are generating a fewer number of false positives (i.e., fewer than 10.0%).
- F3: Existing techniques are more effective towards certain projects and types of APR techniques while less effective to the others (e.g., project *Lang* and *constraint-based* APR techniques for static code features; and project *Closure* and *learning-based* APR techniques for dynamic ones).

F4: Existing techniques are highly complementary to each other. A single technique can only detect at most 53.5% of the overfitting patches while 93.3% of them can be detected by at least one technique when the oracle information is available.

Based on our findings, we designed an integration strategy to first integrate different static code features via leveraging machine-learning models, and then combine the learned model with other APCA techniques by the *majority voting* strategy. Our experiments show that the strategy can enhance the performance of existing APCA techniques significantly. Specifically, our strategy can identify 66.5% of the overfitting patches while preserve a high precision of 99.1% with the oracle information. Such a high recall outperforms the current most effective technique by 25.0%.

2 BACKGROUND AND RELATED WORKS

This section presents the background and related works.

2.1 Automated Program Repair Techniques

Generally, APR techniques can be divided into two categories which are *search-based techniques* and *semantic-based techniques*, respectively [39]. Search-based techniques aim to search for candidate patches within a predefined space, with or without templates as the guidance for code transformation [45]. When without templates, the applied techniques (also known as **heuristic-based approach**) leverage genetic programming [38], random search [63], or multi-objective genetic programming [90] to guide the search of correct patches. Researchers also mine fix templates (also known as **template-based approach**) from history of large-scale open-source projects [29, 36] or from static analysis tools [42]. These templates are applied to generate patches, aiming at producing more correct patches [27, 32, 41, 43, 77]. Semantic-based techniques (also known as **constraint-based approach**) synthesize a patch directly using semantic information via symbolic execution and constraint solving [16, 34, 56, 58, 82, 83]. These approaches usually focus on single conditional statements or assignment statements [56, 58, 83]. Recently, approaches have been proposed to generate patches directly via using deep learning models (e.g., SequenceR [10]), which is denoted as **learning-based approach** [10, 45, 65].

2.2 The Overfitting Problem

Traditionally, a patch is considered as correct if it can pass all the test cases [38, 63, 75]. Qi et al. [64] first investigated the quality of automated generated patches. Long et al. [48] first pointed out that conventional criterion to examine patch correctness is questionable since test suites in practice are usually inadequate to guarantee the correctness of the generated patches. As a result, those generated patches that pass all the tests (also known as *plausible patches*) may fix the bug incorrectly; not fix the bug completely or break some intended functionalities, and thus become *overfitting patches* [79, 89]. After that, researchers begin to adopt plausibility (i.e., how many plausible patches an APR tool can generate) and correctness (i.e., how many generated plausible patches are really correct) as metrics for assessing the repairability of APR tools [9, 23, 25, 27, 32, 42, 43, 46, 77]. Meanwhile, an increasing number of studies aimed at detecting overfitting patches have been proposed [70, 72, 79, 81, 85, 86], including `DiffTGen`, `PATCH-SIM` and so on. Recently, Yu et al. [89] introduced a method to classify overfitting patches into two categorizations (i.e., *incomplete fixing* and *regression introduction*). However, their method can only be applied to those overfitting patches that can be detected by the generated test cases. Consequently, it cannot be applied to the whole set of patches included in this study. As a result, we do not analyze the performance (i.e., *precision* and *recall* as we will introduce in Section 3.3) of different APCA techniques from this perspective.

Table 1: Selected Techniques in this Study.

	Oracle Required	No Oracle Required
Dynamic	`Evosuite` [19], `Randoop` [59], `DiffTGen` [79], `Daikon` [18]	`PATCH-SIM` [81], `E-PATCH-SIM`, `R-Opad` [86], `E-Opad` [86]
Static	⊗	`ssFix`† [80], `CapGen`† [77], `Anti-patterns` [70], `S3`† [34]

† We use `ssFix`, `CapGen`, and `S3` to denote the heuristics based on the corresponding static features.

2.3 Empirical Studies in APR

In recent years, plenty of empirical studies have been conducted concerning different aspects of APR [15, 41, 45, 73]. For instance, Liu et al. found that Fault Localization (FL) strategies [76, 78] utilized by different APR techniques are diverse and the FL results can significantly influence the repair results [41]. Long et al. investigated the search space of repair tools and revealed that correct patches are sparse while the overfitting patches are much more abundant [48], which is further confirmed by [45]. Durieux et al. [15] and Wang et al. [73] focused on benchmark overfitting problem and reached the conclusion that more bugs should be considered when evaluating APR techniques' performance. Recently, Lou et al. explored the idea of *unified debugging* to combine fault localization and program repair in the other direction to boost the performance of both areas[5, 49].

3 STUDY DESIGN

This section presents the design details of this empirical study.

3.1 APCA Techniques Selection

Our study selects all the state-of-the-art techniques targeting assessing patch correctness of Java program. This study focuses on Java since it is the most targeted language in the community of program repair. Furthermore, there is a wide range of APR tools that have been evaluated in real-world Java programs, providing on-hand patches for our study. Specifically, we consider the living review of APR by Monperrus [57] to identify these techniques.

3.1.1 Inclusion. Overall, our study takes totally 9 APCA techniques and 3 heuristics based on 8 static code features into consideration, which can be classified from two aspects as mentioned in Introduction. First, it can be categorized by whether it requires the oracle patch. This corresponds to two different application scenarios: those do not require oracles can be integrated into **patch generation** process and thus help to increase the precision of APR tools while those require oracles are usually used for **patch evaluation** that is to help assess the effectiveness of APR techniques. Second, it can be either dynamic or static, which is differentiated by whether it needs to execute test cases. Table 1 lists the categorized selected techniques and the following presents the detail of each of them.

Simple Test Case Generation: The intuition of this method is straightforward: since most overfitting patches are generated due to the inadequacy of test suites provided by real-world programs [48], researchers proposed to utilize automated test generation tools to generate independent test suites based on the oracle program to examine whether patches are overfitting [35, 68, 69]. If a plausible patch fails in any of these test cases, it is detected as overfitting. Following the previous studies [66, 88], in our experiment, we select Randoop [59] and Evosuite [19] as the test generation tools since they are widely-used in software testing tasks [20, 21, 81].

DiffTGen: DiffTGen is a tool that identifies overfitting patches through test case generation [79]. The tool employs an external test generator (i.e., Evosuite) to generate test input which is designed to uncover the syntactic differences between the patched and the original buggy program (note that this is the difference between this tool and *simple test case generation* where the tests are generated randomly). To achieve so, DiffTGen creates an extended version of the patched program with dummy statements inserted as the coverage goals to advocate test generator. When executing the generated test inputs on the buggy and the patched programs, if the output of the patch is not the same with that of the oracle, it is regarded as overfitting.

Daikon: Some recent studies concentrate on applying program invariant to APR tasks [8, 14, 85]. Specially, Yang et al. focus on the impacts on program runtime behaviors from different patches [85]. They found that a large amount (92/96) of overfitting patches will expose different runtime behaviors (captured by Daikon [18], an invariant generation tool) compared with their corresponding correct versions. Based on their findings, in this study, we adopt a simple heuristic that is to see if the inferred invariant of a generated patch is different from that of the oracle program. If difference exists, we then consider it as overfitting. In the rest of this paper, we use Daikon to represent this method.

Opad: Opad uses fuzzing testing to generate new test cases based on the buggy program, it then uses two predetermined oracles that patches should not introduce new crash or memory-safety problems to detect overfitting patches [86]. To apply it on Java, we adopt the method provided by a recent study [81] that is to uniformly detect whether a patch introduces any new runtime exception on test runs. Note that the original fuzz technique does not work on Java programs. As a result, in this study, we use Randoop and Evosuite to generate test cases on the buggy programs and denote them as R-Opad and E-Opad respectively.

PATCH-SIM: PATCH-SIM is a similarity-based patch validation technique [81] which does not require the oracle information. It first utilizes a test generation tool (Randoop in the original study) to generate new test inputs. It then automatically approximates without the oracle under the hypothesis that tests with similar executions are likely to have the same results. Finally, it uses the enhanced test suite to assess patch correctness considering that a correct patch may behave similarly on passing tests while differently on failing tests compared with the buggy program. In our study, to better explore the performance of this technique, we also implemented another version of this tool by replacing the adopted Randoop with Evosuite for test generation. We use E-PATCH-SIM to represent our Evosuite-based PATCH-SIM.

Anti-patterns: Anti-patterns is originally designed for C language [70]. The authors defined seven categories of program transformation (for details, cf. Table 1 in [70]). To apply it on Java, we follow the strategy adopted by a recent study [81] that is if the code transformation in the patch falls into any category, it is considered as overfitting.

Static Code Features: Many studies have proposed to leverage static code features to prioritize correct patches over overfitting ones [34, 77, 80] and a recent study [2] demonstrates the effectiveness of these features. For instance, ssFix [80] proposed to utilize the *token-based syntax representation* of code to identify syntax-related code fragments with the aim to generate correct patches. S3 proposed six features to measure the syntactic and semantic distance between a candidate solution and the original buggy code [34], and then leveraged such features to prioritize and identify correct patches. These features are named as *AST differencing*, *cosine similarity*, *locality of variables and constants*, *model counting*, *output coverage* and *anti-patterns*. CapGen proposed three context-aware models to prioritize correct patches over overfitting ones, which are the genealogy model, variable model, and dependency model respectively [77]. Although such features are often used to prioritize overfitting patches during patch generation, we still include them in this study with the aim to investigate patch correctness assessment from the view of static features. Note that for S3, the proposed *model counting* can only be applied to Boolean expressions, and *output coverage* can only be applied to program-by-examples based APR. Therefore, they cannot be generalized to all the patches generated by a wide range of APR techniques. Besides, Anti-patterns is used as a stand-alone technique in this study. As a result, we exclude those features for S3 in this study, and Table 2 displays the details of the selected eight features in total. We follow the original studies [34, 77, 80] to compute the values for each feature, and we do not list the formulas in detail due to page limit.

3.1.2 *Exclusion.*

In our study, we also discard some methods that have been exploited by previous studies.

We note that researchers also utilize AgitarOne [1], another test generation tool which is reported to be able to achieve 80% code coverage, to generate tests [66]. We do not take it into consideration since it is a commercial product. KATCH [53] and KLEE [7] are both test generators which leverage symbolic execution and can achieve high coverage. We discard them since they only support C language currently while we focus on Java. UnSatGuided is a method that utilizes test case generation to alleviate overfitting in test suite based program repair [89]. We discard this method since that it only works for synthesis-based repair techniques such as Nopol, and thus cannot be generalized to a wide range of APR techniques selected in this study. A recent study also utilizes *code embedding* technique to identify correct patches [13]. However, it requires tremendous efforts to train the embedding model and thus is discarded in this study. Besides, we note that Ye et al. proposed to use 4,199 static code features to identify overfitting patches recently [87]. However, their tool and data is not publicly available. Besides, their proposed features are atomic ones which are encoded at the level of AST while those selected in this study are high-level features used to encode code syntax and semantics. Therefore, we exclude this method in our empirical study. We also note there are many

Table 2: Static Code Features Used to Prioritize Correct Patches Over Overfitting Ones

Short Name	Metric	Source	Description
TokenStrct	Structural token similarity	ssFix [80]	The similarity between the two vectors representing the structural tokens obtained from the buggy code chunk and the generated patch.
TokenConpt	Conceptual token similarity	ssFix [80]	The similarity between the two vectors representing the conceptual tokens obtained from the buggy code chunk and the generated patch.
ASTDist	AST Difference	S3 [34]	The number of the AST node changes introduced by the patch.
ASTCosDist	Cosine Similarity Distance	S3 [34]	One minus the cosine similarity between the vectors representing the occurrences of distinct AST node types before and after the patch.
VariableDist	Locality of the variables and constants	S3 [34]	The distance is measured by the Hamming distance between the vectors representing the locations of variables and constants.
VariableSimi	Variable Similarity	CapGen [77]	The similarity between the variables involved in the original buggy code element and the applied patch.
SyntaxSimi	Genealogy Similarity	CapGen [77]	The similarity between the syntactic structures (the ancestor and sibling nodes of the corresponding AST) of the original buggy code elements and the applied patch.
SemanticSimi	Dependency Similarity	CapGen [77]	The similarity between the contextual nodes affected by the buggy code elements and the applied patch with respect to their dependencies.

other test generation tools in the literature such as JCrasher [12] and TestFul [4]. We discard them since they are not the state-of-the-art and many of them are no longer maintained (e.g., TestFul). Our selected subjects (i.e., Randoop and Evosuite) are the most widely-used open-source tools to generate tests [11, 21, 33, 67].

3.2 Patch Selection

This section presents the large-scale patches selected in our study.

3.2.1 Patch Benchmark. In this study, we focus on the patches generated for a widely used benchmark by existing APR techniques which is Defects4j [28]. Specifically, we select all the patches prepared by a recent large-scale study [45] where 16 APR systems are evaluated under the same configuration. To better explore the overfitting problem, we also include 269 patches collected by Ye et al. [88] that were not included in [45], including the ones of JAID [9], SketchFix [25], CapGen [77], SOFix [46], and SequenceR [10]. The following two steps are performed based on the selected patches. First, we removed those patches generated for Mockito, as similarly adopted by [81], since some of our studied subject (e.g., Randoop) cannot generate any valid test for this project. Second, we performed a plausibility check to see whether the selected patches are indeed plausible (i.e., can be compiled and pass all the original test suite). To achieve so, we ran each patch on the original test suites again. In total, we discarded 8 patches after the two steps, 2 generated for Mockito and 6 failed in the plausibility check (we have confirmed this with the authors of [45]). Our patch benchmark is: (1) *large-scale*: this benchmark contains in total 902 patches for correctness assessment, and such a number is around 30% more than the recording number (i.e., [87] contains 713 examined patches for correctness) in the literature. (2) *of high coverage w.r.t APR tools*: this benchmark contains the patches generated by 21 distinct APR tools which can be mainly divided into four categories, namely, *heuristic-based*, *constraint-based*, *template-based*, and *learning-based*, as summarized by a recent study [45]. Table 3 shows the detailed information of these APR tools. (3) *of high coverage w.r.t distinct bugs*: this benchmark contains the patches generated for 202 different bugs in Defects4j, accounting for over half of the bugs in the dataset (202/395). Such a high coverage provides great patch diversity for evaluation. Although the learning-based category contains only SequenceR, it contains 73 patches in total, which is almost the largest number for a single APR tool.

3.2.2 Patch Sanity Check. The precise oracle information of the selected patches (i.e., whether a patch is overfitting or correct) is critical in fairly comparing the effectiveness of APCA techniques. The label information of the aforementioned collected patches is annotated by the associated researchers manually. However, as

Table 3: Covered APR Tools in Our Benchmark.

Category	APR Tools for Java Programs
Heuristic-based	jGenProg [54], jKali [54], jMutRepair [54], SimFix [27], ARJA [90], GenProg-A [90], Kali-A [90], RSRepair-A [90], CapGen [77].
Constraint-based	DynaMoth [16], Nopol [83], ACS [82], Cardumen [55], JAID [9], SketchFix [25].
Template-based	kPAR [41], FixMiner [32], AVATAR [42], TBar [43], SOFix [46].
Learning-based	SequenceR [10].

previous studies have pointed out [33, 88], author annotation may produce wrong labels due to subjectivity (i.e., assessing an overfitting patch as a correct one). To reduce such bias in our study, we further performed a sanity check to examine the correctness of the collected patches. To achieve such a goal, we followed the strategies adopted by a recent study [88]. Specifically, we adopted Randoop and Evosuite to generate extra test cases automatically based on the oracle program of the bug (the program after applying the developer-provided patch), and then executed such extra tests against the patched version collected in our dataset that are marked as correct by the authors. We then examined if there were any generated test that passed on the oracle program but failed on the patched version. Such cases indicated that the patch annotated as correct by the authors might be actually overfitting. Therefore, we further manually checked each of them, to see whether they are overfitting via understanding the programs. Three authors were involved and the process ended when they reached consensus. A patch is still considered as correct if three authors admit it is correct. Otherwise, we send the patch with the generated tests to the original authors to see if they agree with our judgement. We deem a correct patch in our dataset is actually overfitting if the original authors also confirm with our judgment. In total, our sanity check identified 12 patches that are mistakenly labeled as correct. Due to page limit, we do not analyze the 12 cases in this paper. Details can be found in our project page at **http://doi.org/10.5281/zenodo.3730599**.

After the process of the sanity check, the patches can be precisely classified to correct patches, which are denoted as $P_{correct}$, and overfitting patches, which are denoted as $P_{overfitting}$. In total, there are 654 $P_{overfitting}$ patches and 248 $P_{correct}$ ones.

3.3 Research Questions

Based on the selected APCA techniques and the precisely labeled patches (i.e., $P_{overfitting}$ and $P_{correct}$), we seek to answer the following research questions (RQs) with the aim to investigate and enhance the effectiveness of existing assessment techniques:

(1) **RQ1: How effective are existing static code features in prioritizing correct patches?** We first systematically investigate the effectiveness of each individual static code feature as summarized in Table 2 in prioritizing correct patches over overfitting ones. To answer this question, we compute the values with respect to each of them for all the patches, and then

compare whether the values obtained over the correct patches are significant different from those obtained over the overfitting ones. If significance observed, the designed feature is promising in differentiating correct patches from overfitting ones. Besides, we also investigate whether the effectiveness of existing features are affected by different projects or different types of APR tools. Answering such a question can guide us better apply existing techniques to different domains, and understand towards which direction should existing techniques be improved.

(2) **RQ2: How effective are existing techniques in identifying overfitting patches?** We systematically investigate the performance of the state-of-the-art APCA techniques on detecting overfitting patches, and whether they are complementary to each other. This question is rather essential to provide valuable guidance for the design of future methods. Specifically, we measure the precision and recall to answer this question, which are defined by the following metrics: **True Positive (TP):** An overfitting patch in $P_{overfitting}$ is identified as overfitting. **False Positive (FP):** A correct patch in $P_{correct}$ is identified as overfitting. **False Negative (FN):** An overfitting patch in $P_{overfitting}$ is identified as correct. **True Negative (TN):** A correct patch in $P_{correct}$ is identified as correct.

$$Precision = TP/(TP + FP) \quad (1)$$

$$Recall = TP/(TP + FN) \quad (2)$$

It has been emphasized by recent studies [81, 87, 89] that APCA techniques need to avoid dismissing correct patches. Therefore, we conjecture an APCA technique effective if it can *generate few false positives while preserve a high recall.* As mentioned in Section 3.1, existing APCA techniques can be characterized in different aspects, such as *dynamic* or *static*, *with oracle* or *without oracle*. In this RQ, we will also investigate the effects of such design spaces on the effectiveness of existing techniques.

(3) **RQ3. Can we enhance the effectiveness of existing techniques via integrating static features and dynamic techniques together?** Based on the experimental outputs obtained from the previous RQs, we further seek to investigate whether combining static and dynamic techniques can achieve better performance in identifying overfitting patches.

3.4 Experiment Settings

All the experiments were performed on the same configured servers with Ubuntu 18.04 x64 OS and 16GB memory. The following presents the experimental details for each research question.

3.4.1 RQ1 & RQ2. ***Static Techniques:*** The four static tools are described in Section 3.1.1. Specifically, to apply Anti-patterns on Java, we follow the heuristic adopted by a recent study [81] that is to manually check whether the patches generated by existing APR tools fall into these patterns. If a generated patch breaks any predefined anti-pattern rule, it is considered as overfitting. For the other three approaches, we calculate the value of each individual feature as described in Table 2. Then, for ssFix and S3, the **sum** of the features is used as the final result while the **multiplication** is used for CapGen, as adopted by the original paper [34, 77, 79].

Evosuite & Randoop: We run these tools on the oracle program to generate tests with 30 different seeds with a time limit of 300 seconds by following previous studies [33, 88]. After collecting those test cases, we run them over the oracle program to eliminate the impact of flaky tests [50]. Any test that fails on the oracle program will be removed from our test suite. Following the previous studies [66, 89], we stop this process if all the oracle patches pass the whole test suite for five times consecutively (note that the sanity check introduced in Section 3.2.2 also went through such a check). After this process, we obtain a test suite with most flaky tests being removed and then execute the automated generated patches against it. A patch is considered as overfitting if it fails any of the test case. Note that for 125 bugs in our study, we directly reuse the Evosuite tests generated by Ye et al. [88] since they have already generated tests for the 125 bugs and removed the flaky tests. Details of these bugs can be found in our replication package.

***R-Opad & E-Opad*:** The experimental setting of these two tools is similar to that of Randoop and Evosuite. The difference is that the test cases are generated based on the buggy programs since Opad does not require the oracle information. Instead, when executing the generated tests on the patched programs, a patch is considered as overfitting if it introduces any runtime exception.

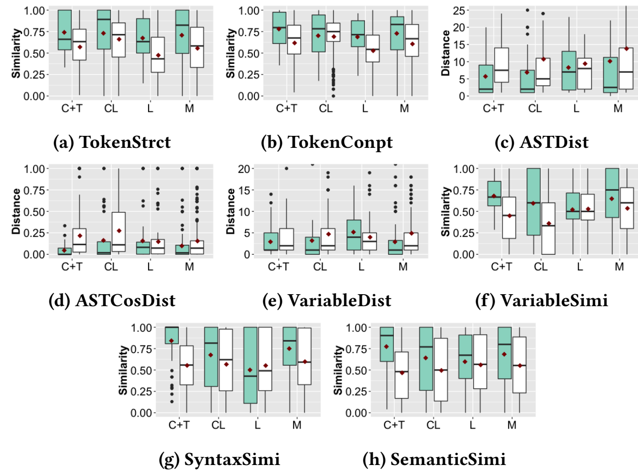
***DiffTGen*:** DiffTGen leverages Evosuite to generate test cases. In our experiment, we execute Evosuite for 30 times and the searching time is set to 60 seconds for each round. We use such settings since it is reported that such a combination is the optimum [80].

***PATCH-SIM & E-PATCH-SIM*:** We note that the performance of our server is lower than that used in the original study. We thus follow the instruction in the project page of [81] that is to increase the timeout limit (3 minutes in original study) to 5 minutes. Both PATCH-SIM and E-PATCH-SIM require two thresholds to classify the generated tests into passing or failing tests (K_t) and the generated patches into correct or overfitting ones (K_p). We adopt the predefined default values as used in [81]. In this study, we do not explore the setting of different thresholds since it has been illustrated in [81] that the impact from different thresholds is limited.

***Daikon*:** Running Daikon is extremely time-consuming [85]. To speed up the process and reduce unnecessary computation, in our experiment, we only infer invariants by executing test classes that contain the failing test cases, as adopted by the previous study [14].

3.4.2 RQ3. To answer RQ3, we first integrate the eight static features as displayed in Table 2 using *learning-to-rank* strategies. Learning to rank techniques train a machine learning model for a ranking task [40], which has been widely used in Information Retrieval tasks. We then integrate the learned model with dynamic techniques via the *majority voting* strategy [61]. To integrate static features, we select six widely-used classification models in our study, including Random Forest [6], Decision Table [93], J48 [60], Naive Bayes [60], Logistic Regression [30], and SMO [62]. These are popular classification models that are wildly used by existing studies [3, 24, 91, 92].

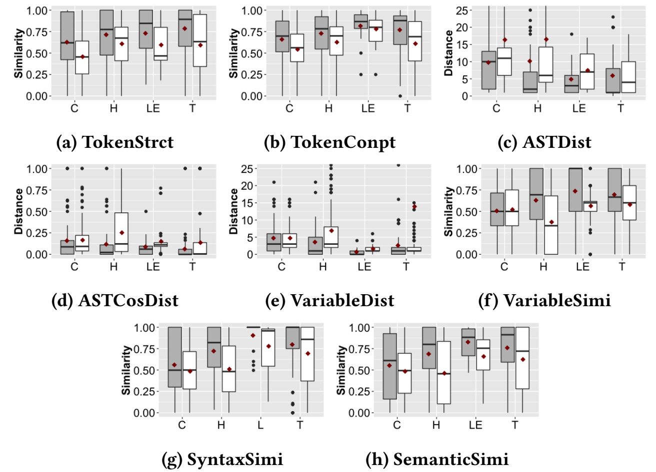
We randomly separate the patch benchmark into 10 folds with identical number of correct and overfitting patches in each fold. We then use 10-fold cross validation [31] to evaluate the performance of each model. The final performance of each model is summed up over the 10 rounds of each training and testing process. Note that our patch benchmark is imbalanced (the number of overfitting patches is around three times as much as that of correct ones), we thus adopt the strategy of *cost-sensitive learning* [17] to increase the loss of generating an FP to 3 times of that of generating an FN. Since



(a) TokenStrct (b) TokenConpt (c) ASTDist (d) ASTCosDist (e) VariableDist (f) VariableSimi (g) SyntaxSimi (h) SemanticSimi

Figure 1: Measurements on Overfitting Patches and Correct Patches for Different Static Code Features Cross Different Projects

The [colored] bar denotes the correct patches while the white bar denotes the overfitting patches. **C+T:** Chart + Time; **CL:** Closure; **L:** Lang; **M:** Math;



(a) TokenStrct (b) TokenConpt (c) ASTDist (d) ASTCosDist (e) VariableDist (f) VariableSimi (g) SyntaxSimi (h) SemanticSimi

Figure 2: Measurements on Overfitting Patches and Correct Patches for Different Static Code Features Cross Different APR Techniques

The [gray] bar denotes the correct patches while the white bar denotes the overfitting patches. **C:** Constraint-based APR; **H:** Heuristic-based APR; **LE:** Learning-based APR; **T:** Template-based APR;

the 10-fold cross validation involves randomness, we repeated this process for 20 times and results suggest that the impact caused by the randomness is limited. For instance, the median of TP generated by *Random Forest* is 580 with a standard deviation of 5.05 while the median of FP is 90.5 with a standard deviation of 3.17. We thus report the result of the first experiment, in this paper.

To combine with dynamic techniques, we select the model with the optimum performance, and integrate it with dynamic ones using the strategy of majority voting. As classified in Table 1, there are two types of dynamic techniques, the one with the oracle information and the other without. However, static code features are designed without the information of oracle. To integrate with those techniques require the oracle information, we also conduct another experiment to leverage the oracle information for static features. Specifically, we re-compute all the feature values based on the generated patch and the oracle patch (in the original experiment, the feature values are computed based on the buggy program and the generated patch), and then adopt the same methodology as described above to integrate all the static code features. Finally, the model learned with the oracle information is combined with the dynamic techniques that require the oracle information.

4 STUDY RESULTS

We now provide the experimental data as well as the key insights distilled from our research questions.

4.1 RQ1: Effectiveness of Static Code Features

Figure 1 shows the measurements over overfitting patches and correct patches with respect to each individual code feature aggregated by different projects. Since the number of patches collected for project Time is not sufficient for statistical difference testing (i.e., only six patches in total), we combined the patches in project Time and Chart, and denoted them as "C+T" in the figures. We can see that for features *TokenStrct*, *TokenConpt*, *VariableSimi*, *SyntaxSimi* and *SemanticSimi*, the values obtained over correct patches are generally higher than those observed over overfitting ones. Such results indicate that correct patches are more likely to preserve high similarities with respect to their syntax and semantics compared with the buggy program. For features *ASTDist*, *ASTCosDist* and *VariableDist*, the values obtained over correct patches are generally lower than those observed over overfitting ones. Such results reveal that correct patches often make a small number of modifications (i.e., as measured by ASTDist), and do not often change the locations of variables/constants (i.e., as measured by VariableDist), as compared to overfitting patches.

To investigate whether such differences observed between correct and overfitting patches are significant, we further performed a one-sided Mann-Whitney U-Test [51] on the results collected from each project, and Table 4 displays the p-values. As we can see from the table, for most of the cases, the differences are significant (i.e., p-value ≤ 0.05), which indicates that existing static code features are effective in differentiating overfitting patches from correct ones. However, the measurements over the Lang projects with respect to six out of the eight features are insignificant. Such a result indicates that existing features are less effective to prioritize patches generated for project Lang. We further investigated the behind reason and found that the buggy locations of this project are mostly concerned with condition expressions, and the overfitting patches are often generated via modifying operators in such expressions. Listing 1 shows some examples of such overfitting patches. Unfortunately, existing code features are unable to capture the characteristics of such differences embedded in operators. As a result, the differences measured over the overfitting patches and those over the correct patches are insignificant.

```
1  // An overfitting patch generated for Lang 22 by jMutRepair
2  - if (Math.abs(u) <= 1 || Math.abs(v) <= 1) {
3  + if (Math.abs(u) <= 1 && Math.abs(v) <= 1) {
4
5  // An overfitting patch generated for Lang 51 by TBar
6  - if (ch == 'y') {
7  + if (ch <= 'y') {
```

Listing 1: Overfitting Patches Generated for the Lang Project

Table 4: P-values over All the Static Code Features

Feature Name	Projects				Types of APR Technique			
	C+T	CL	L	M	C	H	LE	T
TokenStrct	0.0002	0.0165	0.0000	0.0000	0.0000	0.0011	0.0335	0.0000
TokenConpt	0.0000	**0.2003**	0.0001	0.0000	0.0006	0.0004	**0.0645**	0.0000
ASTDist	0.0000	0.0013	**0.5245**	0.0001	0.0116	0.0000	0.0213	**0.0570**
ASTCosDist	0.0000	0.0001	**0.4123**	0.0005	**0.0600**	0.0000	0.0032	0.0049
VariableDist	0.0045	0.0016	**0.9493**	0.0000	**0.2830**	0.0001	0.0001	0.0330
VariableSimi	0.0000	0.0001	**0.4194**	0.0011	**0.6681**	0.0000	0.0067	0.0011
SyntaxSimi	0.0000	0.0098	**0.7648**	0.0000	**0.0703**	0.0000	0.0007	0.0066
SemanticSimi	0.0000	0.0039	**0.3331**	0.0002	**0.0675**	0.0000	0.0045	0.0043

0.0000 denotes the p-value is less than 0.00005

Table 5: Effectiveness of each APCA Technique.

APCA	TP	FP	TN	FN	Precison	Recall
Evosuite	350	3	245	304	99.15%	53.52%
Randoop	221	6	242	433	97.36%	33.79%
DiffTGen†	184	5	232	417	97.35%	30.62%
Daikon†	337	38	166	120	89.87%	73.74%
R-Opad	67	0	248	587	100.00%	10.24%
E-Opad	92	0	248	562	100.00%	14.07%
PATCH-SIM†	249	51	186	392	83.00%	38.85%
E-PATCH-SIM†	166	36	202	477	82.18%	25.82%
Anti-patterns	219	37	211	435	85.55%	33.49%
S3	516	135	113	138	79.26%	78.90%
ssFix	515	138	110	139	78.87%	78.75%
CapGen	506	140	108	148	78.33%	77.37%

The green cell denotes the technique requires the oracle information. The bold name means the technique is dynamic. †For these tools, results might not be generated for certain patches. The reasons for non-result generation are explained in detail in this Section.

We performed similar analysis with respect to different types of APR techniques. Figure 2 shows the results and Table 4 shows the corresponding p-values. As we can see from the figures, similar results can be observed compared with those obtained with respect to different projects. Specifically, existing static code features are generally effective in prioritizing overfitting patches over correct ones with respect to different types of APR techniques, and the differences are mostly significant as shown in Table 4. However, insignificance is frequently observed for those patches generated by *constraint-based* APR techniques. We investigated the behind reason and found that patches in this type often extend the condition of an *if* statement with rare new variables involved as shown in the example displayed in Listing 2. Note that the token list in this patch changes significantly compared with that of the buggy program. Therefore, token-based features are more effective for this type of patches, as shown in Table 4, compared with other features that encode variables or the related semantics.

```
1  // An overfitting patch generated for Math 28 by ACS
2  - } else if(minRatioPositions.size() > 1) {
3  + } else if(minRatioPositions.size() > 1 && !(minRatioPositions.size() > 0))
```

Listing 2: An Overfitting Patch Generated by ACS

The systematic study on static code features reveals that:

(1) existing static code features are generally effective in differentiating overfitting patches over correct ones; (2) these features are less effective towards project Lang and constraint-based APR techniques while more effective with respect to other types of patches.

4.2 RQ2: Effectiveness of APCA Techniques

4.2.1 Evaluation Results. Table 5 shows the detection results of each APCA technique on our patch benchmark. As indicated by the results, among all the 654 overfitting patches, existing dynamic APCA techniques can identify a diverse number of them, ranging from 67 to 350 for different techniques. For instance, Evosuite identified 350 overfitting patches (53.52% of the $P_{overfitting}$) while it also labeled 3 correct patches as overfitting; Daikon identified 337 overfitting patches while it produced 38 false positives. Besides, our two implementations of Opad (i.e., R-Opad and E-Opad) generated no false positives but they detected the least number of overfitting patches. Other dynamic techniques that do not require the oracle information (e.g., E-PATCH-SIM and PATCH-SIM), generated more number of false positives. The behind reasons of such false positives for each technique will be discussed in detail subsequently.

We also investigated the distributions of the overfitting patches identified by different APCA techniques, and the results are shown in Figure 3. Due to the space limit of the Venn figure, we do not include R-Opad and E-Opad in this comparison since they detect the least number of overfitting patches. Actually, these two tools can only uniquely detect 9 and 0 overfitting patches, respectively. As we can see from this figure, only 11 overfitting patches were detected by all the displayed techniques while substantial overfitting patches were detected exclusively by specific techniques. For instance, 48 overfitting patches can be detected by Evosuite or Randoop that cannot be detected by other techniques. In total, by combining all the dynamic APCA techniques plus Anti-patterns, we can detect 610 unique overfitting patches, accounting for 93.3% of the overfitting patches in our benchmark. Such results indicate that considering multiple factors when designing APCA techniques can achieve better performance.

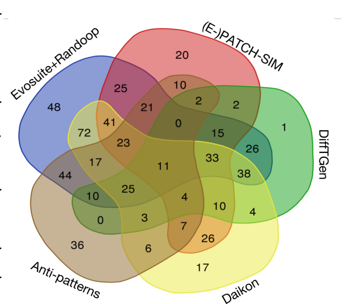


Figure 3: Overfitting Patches Dist.

Heuristics based on static code features are not designed as a standalone technique to identify overfitting patches directly. However, we also list the results in Table 5 to compare with other techniques. Specifically, we calculated the value for each patch, generated by ssFix, CapGen, and S3 respectively, as introduced in Section 3.1.1 and then prioritized all the patches based on the obtained values and their ranking strategies. The top 248 patches are classified as correct patches and the remaining ones are regarded as overfitting ones (The number of 248 is selected since our prepared patch benchmark is unbalanced which only contains 248 correct patches). As observed in Table 5, heuristics based on static code features identified more number of overfitting patches in general compared with dynamic ones. However, they are less precise as well since they generated more number of false positives. For instance, although CapGen classified 506 patches as overfitting correctly, it also labeled 140 correct patches as overfitting ones. Applying such techniques will cause significant destructive effects (i.e., dismissing a number of correct patches), in which case the effectiveness of APR techniques will be significantly under-estimated.

4.2.2 Analysis of the Results. We further performed a deeper analysis for the above results, and the following presents our findings.

• [***False positives cases***] We carefully analyzed the false positives generated by APCA techniques and identified the following reasons. For Evosuite, it produced 3 false positives (i.e., the patches for Lang-7 generated by ACS, kPAR, and TBar) which are caused by the fact that the generated tests have broken the target program's preconditions. The oracle patch for Lang-7 adds a conditional statement to deal with unexpected input in function *createBigDecimal()*. These three patches performed the correct modification as the oracle patch but in a different location (in the function *createNumber()*). There exists a precondition of this program that *createBigDecimal()* will only be called by *createNumber()* and that is why these patches are correct. Evosuite generated tests that violate this precondition by calling the function *createBigDecimal()* directly. Consequently, the tests failed on the patched program and these patches are considered as overfitting. For Randoop, it generated tests that check the version for bug Closure-115 as shown in Listing 3. These tests only pass on the oracle program and thus, six generated correct patches failed on them.

```
1  // A test case generated by Randoop
2  String str0 = Compiler.getReleaseVersion();
3  assertTrue("'" + str0 + "' != '" + "D4J_Closure_115_FIXED_VERSION" + "'",
         str0.equals("D4J_Closure_115_FIXED_VERSION"));
```

Listing 3: A Test Case Generated by Randoop

For DiffTGen, besides generating the same three false positives as Evosuite, it also misclassified another two correct patches. An example is shown in Listing 4. ACS generated a patch which is semantic equivalent to the oracle patch by throwing the same exception. However, it did not synthesis the thrown message (i.e., "Object must implement Comparable"). DiffTGen generated a test that checks whether the thrown messages of these two programs are identical. As a result, this correct patch is mislabeled.

For Daikon, it in total generated 38 false positives since the rules adopted by Daikon to affirm identical invariant is extremely strict. Note that Daikon infers invariant at every entry and exit points of the functions, and if any differences is observed, Daikon will label it as an overfitting patch. Specifically, among the 38 false positive cases, 30 of them concern different exit points of functions and 7 of them use different program elements to finish the same target. Note that if a function have multiple exit points, Daikon will print the line number of each exit point in its output. Considering the example in Listing 4, *line 6* and *line 12* are the exit points of the oracle and patched programs. Their line numbers with respect to the whole program are different (the former is 113 while the latter is 111). This leads to the differences between the outputs of Daikon, and thus this correct patch is mislabeled as overfitting. Another uncommon case is patch for Lang-55 generated by *Jaid*. The program calls *System.currentTimeMillis()* during execution which generates different timestamps, thus causing different invariants. Such results indicate that future work leveraging invariants to identify overfitting patches can consider to weaken the rules to compare identical invariants, and also consider the characteristics of the inferred invariants and treat them differently when classifying patches.

```
1  // Oracle patch for Math 89
2    public void addValue(Object v) {
3  +  if (v instanceof Comparable<?>){
4       addValue((Comparable<?>) v);
5  +  } else {
6  +   throw new IllegalArgumentException("Object must implement Comparable");
7  +  }
8
9  // A correct patch for Math 89 by ACS
10   public void addValue(Object v) {
11 +  if (!(v instanceof Comparable<?>)){
12 +   throw new IllegalArgumentException();}
13    addValue((Comparable<?>) v);
14 // Test case generated by DiffTGen
15 assertEquals(
16   "(E)0,(C)org.apache.commons.math.stat.Frequency,addValue(Object)0,(I)0",
17   "java.lang.IllegalArgumentException: Object must implement Comparable",
18   ((Throwable) target_obj_7au3e).toString());
```

Listing 4: An Example of False Positive of DiffTGen and Daikon

For PATCH-SIM and E-PATCH-SIM, they generated 87 false positives in total. We observed that these cases are majorly caused by the misclassification of the generated tests due to the complexity of the original developer-provided test suite. Listing 5 shows a concrete example. The correct patch adds a conditional statement to deal with the unexpected input *null*. However, the original failing test is complex, which feeds the function *getRangeAxisIndex* with diverse inputs that cover both normal and unexpected inputs. Such a complex test leads to a complex path spectrum of *getRangeAxisIndex* during execution. PATCH-SIM generated a test case which calls this function only with the input *null*, leading to an extremely limited path spectrum on this method. As a result, this generated test is misclassified as a passing test since the execution trace is divergent to that of the original failing test. When determining the correctness of this correct patch, the execution trace of this test on the buggy program in this function is lines 6, 7, and a return statement, completely different from that of the patched program which is lines 3 and 4. Consequently, this test leads to an extremely high value of A_p (the distance of a passing test) which is 0.53, thus misclassifying the patch as overfitting (cf. Section 4.4.2 in [81]). Similar patterns have been observed for most of the other FP cases, that is the distance between the path spectrum obtained via executing a passing test against a correct patch and that obtained against the buggy program is significant different. The behind reason is that the logic of the developer-provided test suite is complex. Therefore, utilizing *test case purification* technique [27, 84] (i.e., separating those normal inputs from abnormal ones in the example) is promising to enhance the accuracy to classify the generated tests, thus to enhance the effectiveness of PATCH-SIM.

```
1  // A correct patch for Chart 19 by AVATAR
2    public int getRangeAxisIndex(ValueAxis axis) {
3  +   if (axis == null) {
4  +     throw new IllegalArgumentException();
5  +   }
6      int result = this.rangeAxes.indexOf(axis);
7      if (result < 0) {
8  // A test case classified as passing test
9    int int18 = categoryPlot7.getRangeAxisIndex(null);
10   assertTrue(int18 == 0);
```

Listing 5: An Example of False Positive of PATCH-SIM

For Anti-patterns, we further dissected the effectiveness of each rule adopted by it, including the detected number of overfitting patches and the number of false positive cases. We find that only five rules are effective in detecting existing overfitting patches in the Java language and all these five rules also produce false positive cases, which proves that applying these patterns will inevitably cause certain destructive effects (i.e., dismissing a number of correct patches) as revealed by another study [22].

Table 6: Performances of Existing APCA Techniques with Respect to Different Projects and Different APR Techniques

APCA Technique	Precision								Recall							
	C+T	CL	L	M	H	T	C	LE	C+T	CL	L	M	H	T	C	LE
Evosuite	100.00%	100.00%	95.71%	100.00%	100.00%	97.73%	98.75%	100.00%	86.79%	20.51%	63.81%	60.89%	51.55%	53.75%	68.10%	33.93%
Randoop	100.00%	91.04%	100.00%	100.00%	96.06%	98.18%	100.00%	100.00%	38.68%	31.28%	41.90%	30.24%	37.89%	33.75%	37.07%	3.57%
DiffTGen	97.83%	100.00%	91.67%	99.04%	98.89%	96.08%	94.29%	100.00%	43.69%	1.79%	35.87%	43.28%	30.58%	34.27%	28.70%	25.00%
Daikon	92.93%	NA	82.29%	92.22%	95.03%	93.52%	78.10%	100.00%	86.79%	NA	75.24%	67.48%	79.69%	76.52%	75.23%	4.17%
R-Opad	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	18.87%	9.23%	11.43%	6.85%	9.32%	8.75%	8.62%	23.21%
E-Opad	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	13.21%	5.13%	12.38%	22.18%	13.98%	16.88%	10.34%	14.29%
Anti-patterns	95.45%	89.32%	63.64%	84.21%	93.98%	69.57%	68.25%	100.00%	39.62%	47.18%	20.00%	25.81%	48.45%	10.00%	37.07%	7.14%
PATCH-SIM	72.92%	75.00%	80.00%	86.17%	87.86%	73.40%	69.84%	68.42%	66.04%	22.22%	54.90%	33.20%	39.05%	44.23%	38.26%	23.64%
E-PATCH-SIM	80.00%	71.93%	62.96%	80.00%	83.00%	68.25%	73.68%	66.67%	52.83%	21.47%	16.83%	21.22%	26.27%	27.39%	24.35%	21.82%

The darker blue of the cell, the higher value; The darker orange of the cell, the lower value; We did not compare with static code features since they have been studied in RQ1.

Different Project: C+T: Chart + Time; **CL:** Closure; **L:** Lang; **M:** Math. **Different Techniques: C:** Constraint-based APR; **H:** Heuristic-based APR; **LE:** Learning-based APR; **T:** Template-based APR;

• [***Results in contradict with previous studies***] We note that our experiment reports much more false positives of PATCH-SIM than the original study [81]. We carefully checked our results and confirmed that for the set of 24 correct patches that are selected in both the two studies, only the classification result of one patch is different. Considering the randomization of Randoop, we conjecture it is reasonable. Therefore, our study exposes the threats to external validity of the previous study [81] using a large-scale benchmark, as mentioned in [22]. We also note that our experiment reports much more true positives of two implementations of Opad (67 and 92, respectively) than the number reported in [81]. The reason is that the test suite used in our experiment contains hundreds of tests for a patch rather than at most 50 as adopted in [81].

• [***Reasons for non-result generation***] Certain dynamic APCA tools did not generate results for certain patches: for Daikon, it cannot generate invariants for *Closure* programs (also reported in [85], 241 cases in total); for PATCH-SIM and E-PATCH-SIM, they run out of heap space when a large amount of tests cover the patched method (also reported in [22], 24 cases for PATCH-SIM and 21 cases for E-PATCH-SIM); for DiffTGen, it does not work if the oracle patch deletes a method (since it needs an oracle method to check the correctness of the patch) or if the types and values of the inputs and outputs of the patched method cannot be obtained (since it needs these information to perform the check, cf. Section 3.3.1 in [79]). There are 64 cases in total and we have confirmed these limitations with the authors of DiffTGen.

• [***Performance on different APR types & Defects4j projects***] Table 6 shows the results of existing APCA techniques in terms of precision and recall, in particular, separated by different types of APR techniques and different projects. In terms of precision, existing APCA techniques are generally achieving high performances cross different projects and APR tools. Specifically, Anti-patterns, PATCH-SIM and E-PATCH-SIM achieve relatively lower precision values, which mostly vary from 60% to 90%. Other techniques achieve better performances, which are mostly greater than 90%, and for 60.4% of the cases, the precision achieved is 100.0%.

In terms of recall, the performance of existing APCA techniques diverges a lot cross different projects and APR techniques. Specifically, the recalls achieved over projects Chart and Time are generally higher than that over the other projects while the recall of project Closure is the lowest. It is caused by the fact that the test suites on Closure often achieve low coverage [66] while dynamic techniques rely heavily on the quality of the generated tests. The recalls achieved over the *learning-based APR* technique (i.e., SequenceR) are generally lower than that over the other types of APR techniques. Especially, we found three APCA techniques achieved rather low recalls on this type of patches (i.e., Randoop, Daikon, and Anti-patterns). We observed the following reasons: for Randoop, the test suite generated for the bugs of this type of patches generally achieve low coverage; for Anti-patterns, learning-based APR seldom generate patches via code transformation that can be captured by the patterns; for Daikon, the variables modified by the patches are generally not the return values, in which case their changes are rarely captured by the inferred invariants. Such results indicates that existing techniques are comparatively less effective in identifying overfitting patches towards learning-based APR techniques. Overall our systematic study of the effectiveness of existing APCA techniques reveals that:

(1) most of the dynamic APCA techniques will label correct patches as overfitting while those with the oracle information are generating a fewer number of false positives (i.e., fewer than 10.0%); (2) existing APCA techniques are highly complementary to each other since a single technique can only detect at most 53.5% overfitting patches while 93.3% of the overfitting ones can be detected by at least one technique with the oracle; (3) heuristics based on static code features can achieve higher recalls but are less precise by generating more number of false positives; (4) existing APCA techniques are less effective towards project Closure and learning-based APR techniques while more effective with respect to other types of patches.

4.3 RQ3: Learning & Integration

Previous findings reveal that *static features can achieve high recall while dynamic techniques are more effective towards precision* and *existing techniques are highly complementary to each other*. Therefore, it motivates us to integrate existing techniques together to take the advantage of each side's merits.

4.3.1 Integrating Static Code Features. Table 7 shows the results of our method to integrate all the static features via learning through six machine learning models. From the results, *Random Forest* is the most effective model since it achieves the highest recall while still maintaining a relatively high precision for the two experimental settings. In particular, for the setting without the oracle information, it achieves a high recall of 89.14% while preserving a high precision of 87.01%. We find that our model achieves better performance than a single tool (e.g., *Random Forest* generates more TP with

Table 7: Effectiveness of Each Training Model based on the Eight Static Features with and without the Oracle Information

	Decision Table		J48		Logistic Regression		NaiveBayes		Random Forest		SMO	
	without	with	without	with	without	with	without	with	without	with	without	with
TP	496	588	506	584	354	501	130	318	583	623	447	599
FP	85	84	84	57	88	64	31	46	87	64	85	71
Precision	85.37%	87.50%	85.76%	**91.11%**	80.09%	88.67%	80.75%	87.36%	**87.01%**	90.68%	84.02%	89.40%
Recall	75.84%	89.91%	77.37%	89.30%	54.13%	76.61%	19.88%	48.62%	**89.14%**	**95.26%**	68.35%	91.59%

The optimum performance is displayed **in bold**. "without" means the oracle patch is not available. "with" means the oracle is available. The same as Table 8.

Table 8: Integration Results with and without the Oracle

	Strategy	TP	FP	Precision	Recall
without	PATCH-SIM	249	51	83.00%	38.85%
	Anti-patterns	219	37	85.55%	33.49%
	Integration with the Learned Model	**343**	**30**	**91.96%**	**52.45%**
	PATCH-SIM + E-PATCH-SIM + Anti-patterns	182	38	82.73%	27.83%
with	Evosuite	350	**3**	**99.15%**	53.52%
	Randoop	221	6	97.36%	33.79%
	Integration with the Learned Model	**435**	4	99.10%	**66.51%**
	Evosuite + Randoop + Daikon	295	**3**	98.99%	45.11%

fewer FP than `ssFix`, `CapGen`, and `S3` as shown in Table 5). For instance, compared with `S3`, the most effective one among the three, the precision and recall have been improved by 9.78% and 13.0% respectively. This suggests that integrating diverse static features via learning to identify overfitting patches is a promising future direction. We also find that the oracle information can boost the effectiveness of all the models via achieving both a higher *precision* and *recall*. For instance, for the model of *Random Forest*, the number of TP has been increased by 40 while the number of FP has been reduced by 23. Such a result indicates using such models to facilitate the evaluation of APR techniques is promising.

4.3.2 *Integrating with Existing Techniques.*

We choose the most effective model (*Random Forest*) to integrate with dynamic techniques and `Anti-patterns`. Our intuition is that they can guarantee the precision while the trained model can help enhance the recall. As is well known, dynamic methods can be extremely time-consuming [88], and thus we only consider the top two most effective techniques among dynamic ones and `Anti-patterns` for integration currently. Specifically, for the scenario with the oracle information, we integrate our trained model (with the oracle information) with `Evosuite` and `Randoop`. For the scenario without the oracle information, we integrate our trained model (without the oracle information) with `PATCH-SIM` and `Anti-patterns`. We adopt the *majority voting* strategy that is a patch is considered as overfitting if it has been labeled as overfitting by at least two out of our considered techniques. Table 8 shows the results, which indicates that the effectiveness of existing APCA techniques can be significantly enhanced via integration. Specifically, for the scenario without the oracle information, while preserving a low FP of 30, the TP can be improved from 249 to 343, thus achieving a higher recall (52.45% vs. 38.85%). For the scenario with the oracle information, while preserving a high precision of 99.10%, the recall can be improved from 53.52% to 66.51%. To show the contributions of the learned model in the integration strategy, we also compared with the results obtained by integrating the three most effective existing techniques via the majority voting. The results are shown in the last row of each experimental setting which indicate that without the learning model based on static code features, the effectiveness of identifying overfitting patches is greatly compromised. For instance, if we integrate `Evosuite`, `Randoop`, and `Daikon`, we can only achieve a recall of 45.11%, even lower than a single method like `Evosuite`. Such results reflect the usefulness of static code features. Our integration strategy reveals that:

> *(1) combining static features via learning significantly outperforms existing heuristics based on static code features and thus is a promising direction for both patch generation and patch evaluation; (2) integrating static features with existing techniques can take the advantage of each side, thus achieving a higher recall while preserving a high precision. Therefore, it is a future direction worth exploring.*

5 THREATS TO VALIDITY

External validity. Our study only considers the patches generated by Java APCA techniques on the Defects4J benchmark. Thus, all findings might be only valid for this configuration. Nevertheless, this threat is mitigated by the fact that we use a wide range of state-of-the-art APCA techniques and a most comprehensive patch benchmark so far.

Internal validity. It is error-prone to perform such a large scale study and some of our findings may face the threats from the way we performed the experiments. We mitigate this by re-checking the process of our experiment for many times and identifying the behind explanation for each result. Besides, we have reported some of our results to authors of [45, 79, 82] and obtained positive feedback.

Construct validity. The parameters involved in this study may cast effects for the results. We mitigate this by strictly following the previous studies. For instance, we run *simple test case generation* tools for 30 seeds by following [88, 89] and adopt the default values of K_t and K_p from [81] for `PATCH-SIM` and `E-PATCH-SIM`.

6 CONCLUSION

This paper performed a large-scale study on the effectiveness of 9 state-of-the-art APCA techniques and 3 heuristics based on 8 static code features, based on the largest patch benchmark so far. Effectiveness is evaluated and compared with respect to *precision* and *recall*. Our study dissects the pros and cons of existing approaches as well as points out a potential direction by integrating static features with existing methods.

Artefacts: All data in this study are publicly available at:

http://doi.org/10.5281/zenodo.3730599.

ACKNOWLEDGMENTS

The authors thank He Ye from KTH Royal Institution and Qi Xin from Georgia Institute of Technology for their great help in the experiment. This work is supported by the National Natural Science Foundation of China No.61672529 as well as the Fundamental Research Funds for the Central Universities (HUST) No.2020kfyXJJS076.

REFERENCES

[1] 2020. AgitarOne Homepage. http://www.agitar.com/index.html.

[2] Moumita Asad, Kishan Kumar Ganguly, and Kazi Sakib. 2019. Impact Analysis of Syntactic and Semantic Similarities on Patch Prioritization in Automated Program Repair. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 328–332.

[3] Lingfeng Bao, Xin Xia, David Lo, and Gail C Murphy. 2019. A large scale study of long-time contributor prediction for GitHub projects. *IEEE Transactions on Software Engineering* (2019).

[4] Luciano Baresi, Pier Luca Lanzi, and Matteo Miraz. 2010. TestFul: an Evolutionary Test Approach for Java. In *3rd International Conference on Software Testing*. IEEE.

[5] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2020. On the Effectiveness of Unified Debugging: An Extensive Study on 16 Program Repair Systems. In *the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020)*.

[6] Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *International Conference on Software Engineering*.

[7] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. 209–224.

[8] Padraic Cashin, Carianne Martinez, Westley Weimer, and Stephanie Forrest. 2019. Understanding Automatically-Generated Patches Through Symbolic Invariant Differences. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 411–414.

[9] Liushan Chen, Yu Pei, and Carlo A Furia. 2017. Contract-based program repair without the contracts. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 637–647.

[10] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noel Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/tse.2019.2940179

[11] Hangyuan Cheng, Ping Ma, Jingxuan Zhang, and Jifeng Xuan. 2020. Can This Fault Be Detected by Automated Test Generation: A Preliminary Study. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*. 9–17.

[12] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: an automatic robustness tester for Java. *Software Prac. Experience* 34, 11 (2004), 1025–1050.

[13] Viktor Csuvik, Dániel Horváth, Ferenc Horváth, and László Vidács. 2020. Utilizing Source Code Embeddings to Identify Correct Patches. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*. 18–25.

[14] Zhen Yu Ding, Yiwei Lyu, Christopher Timperley, and Claire Le Goues. 2019. Leveraging program invariants to promote population diversity in search-based automatic program repair. In *2019 IEEE/ACM International Workshop on Genetic Improvement (GI)*. IEEE, 2–9.

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

[16] Thomas Durieux and Martin Monperrus. 2016. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th IEEE/ACM International Workshop in Automation of Software Test*. IEEE, 85–91.

[17] Charles Elkan. 2001. The foundations of cost-sensitive learning. In *International joint conference on artificial intelligence*. 973–978.

[18] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (Feb. 2001), 99–123.

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

[20] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[21] Gordon Fraser, Andrea Arcuri, and Phil McMinn. 2015. A memetic algorithm for whole test suite generation. *Journal of Systems and Software* 103 (2015), 311–327.

[22] Ali Ghanbari. 2019. Validation of Automatically Generated Patches: An Appetizer. arXiv:1912.00117

[23] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 19–30.

[24] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. 2015. Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 789–800.

[25] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 12–23.

[26] Jiajun Jiang, Yingfei Xiong, and Xin Xia. 2019. A manual inspection of Defects4J bugs and its implications for automatic program repair. *Science China Information Sciences* 62, 10 (2019), 200102.

[27] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 298–309.

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

[29] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 802–811.

[30] David G Kleinbaum, K Dietz, M Gail, Mitchel Klein, and Mitchell Klein. 2002. *Logistic regression*. Springer.

[31] Ron Kohavi. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Fourteenth International Joint Conference on Artificial Intelligence*. Montreal, Canada, 1137–1145.

[32] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2018. Fixminer: Mining relevant fix patterns for automated program repair. *arXiv preprint arXiv:1810.01791* (2018).

[33] Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On reliability of patch correctness assessment. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 524–535.

[34] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 593–604.

[35] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2017. Empirical Study on Synthesis Engines for Semantics-Based Program Repair. In *Proceedings of the 33th IEEE International Conference on Software Maintenance*. IEEE.

[36] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 213–224.

[37] Xuan Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. *Empirical Software Engineering* 23, 5 (2018), 3007–3033.

[38] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[39] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (2019), 56–65. https://doi.org/10.1145/3318162

[40] Hang Li. 2011. A short introduction to learning to rank. *IEICE TRANSACTIONS on Information and Systems* 94, 10 (2011), 1854–1862.

[41] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques. Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification*. 102–113. https://doi.org/10.1109/ICST.2019.00020

[42] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 456–467.

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

[44] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. 2018. LSRepair: Live Search of Fix Ingredients for Automated Program Repair. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference*. 658–662. https://doi.org/10.1109/APSEC.2018.00085

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

[46] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 118–129.

[47] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 166–178.

[48] Fan Long and Martin Rinard. 2016. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 702–713. https://doi.org/10.1145/

2884781.2884872

[49] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can Automated Program Repair Refine Fault Localization? A Unified Debugging Approach. In *Proceedings of the 29th ACM International Symposium on Software Testing and Analysis (ISSTA 2020)*.

[50] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 643–653.

[51] Henry B Mann and Donald R. Whitney. 1947. On a Test of Whether One of Two Random Variables Is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50–60. https://doi.org/10.1214/aoms/1177730491

[52] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 269–278.

[53] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: high-coverage testing of software patches. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*.

[54] Matias Martinez and Martin Monperrus. 2016. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 441–444.

[55] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: the Cardumen Mode of Astor. In *Proceedings of the 10th International Symposium on Search Based Software Engineering*. Springer, 65–86.

[56] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[57] Martin Monperrus. 2018. *The Living Review on Automated Program Repair*. Technical Report. Technical Report hal-01956501. HAL/archives-ouvertes. fr, HAL/archives.

[58] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 772–781.

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

[60] Tina R. Patil and Swati Sunil Sherekar. 2013. Performance analysis of Naive Bayes and J48 classification algorithm for data classification. *International journal of computer science and applications* 6, 2 (2013), 256–261.

[61] Lionel S Penrose. 1946. The elementary statistics of majority voting. *Journal of the Royal Statistical Society* 109, 1 (1946), 53–57.

[62] John Platt. 1998. Sequential minimal optimization: A fast algorithm for training support vector machines. MSR-TR-98-14 (1998), 21. https://www.microsoft.com/en-us/research/publication/sequential-minimal-optimization-a-fast-algorithm-for-training-support-vector-machines/

[63] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 254–265.

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

[65] Andrew Scott, Johannes Bader, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *arXiv preprint arXiv:1902.06111* (2019).

[66] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *International Conference on Automated Software Engineering (ASE)*.

[67] Inderjeet Singh. 2012. A mapping study of automation support tools for unit testing. In *School of Innovation Design and Engineering*.

[68] Edward K Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 532–543.

[69] Mauricio Soto and Claire Le Goues. 2018. Using a probabilistic model to predict bug fixes. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 221–231.

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

[71] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. 2020. Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM.

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

[73] Shangwen Wang, Ming Wen, Xiaoguang Mao, and Deheng Yang. 2019. Attention please: Consider Mockito when evaluating newly proposed automated program repair techniques. In *Proceedings of the 23rd Evaluation and Assessment on Software Engineering*. ACM, 260–266.

[74] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 356–366.

[75] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE, 364–374.

[76] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2020. Historical Spectrum based Fault Localization. *IEEE Transactions on Software Engineering (TSE)* (2020).

[77] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1–11.

[78] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 262–273.

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

[80] Qi Xin and Steven P. Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 660–670.

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

[82] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*. IEEE, 416–426.

[83] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55.

[84] Jifeng Xuan and Martin Monperrus. 2014. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 52–63. http://doi.acm.org/10.1145/2635868.2635906

[85] Bo Yang and Jinqiu Yang. 2020. Exploring the Differences between Plausible and Correct Patches at Fine-Grained Level. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*. IEEE, 1–8.

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

[87] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. Automated Classification of Overfitting Patches with Statically Extracted Code Features. arXiv:1910.12057

[88] He Ye, Matias Martinez, and Martin Monperrus. 2019. Automated Patch Assessment for Program Repair at Scale. arXiv:1909.13694

[89] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2019. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system. *Empirical Software Engineering* 24, 1 (2019), 33–67.

[90] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering* (2018).

[91] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. 2018. Predictive mutation testing. *IEEE Transactions on Software Engineering* 45, 9 (2018), 898–918.

[92] Tianchi Zhou, Xiaobing Sun, Xin Xia, Bin Li, and Xiang Chen. 2019. Improving defect prediction with deep forest. *Information and Software Technology* 114 (2019), 204–216.

[93] Wojciech Ziarko and Shan Ning. 1997. Machine Learning Through Data Classification and Reduction. *Fundamenta Informaticae* 30, 3 (1997), 373–382.