Identifying Patch Correctness in Test-Based Program Repair

Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, Gang Huang
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University, Beijing, 100871, China
{xiongyf,liuxinyuan,mhzeng,zhanglucs,hg}@pku.edu.cn

ABSTRACT

Test-based automatic program repair has attracted a lot of attention in recent years. However, the test suites in practice are often too weak to guarantee correctness and existing approaches often generate a large number of incorrect patches.

To reduce the number of incorrect patches generated, we propose a novel approach that heuristically determines the correctness of the generated patches. The core idea is to exploit the behavior similarity of test case executions. The passing tests on original and patched programs are likely to behave similarly while the failing tests on original and patched programs are likely to behave differently. Also, if two tests exhibit similar runtime behavior, the two tests are likely to have the same test results. Based on these observations, we generate new test inputs to enhance the test suites and use their behavior similarity to determine patch correctness.

Our approach is evaluated on a dataset consisting of 139 patches generated from existing program repair systems including jGenProg, Nopol, jKali, ACS and HDRepair. Our approach successfully prevented 56.3% of the incorrect patches to be generated, without blocking any correct patches.

ACM Reference Format:
Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, Gang Huang. 2018. Identifying Patch Correctness in Test-Based Program Repair. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180182

1 INTRODUCTION

In the past decades, a large number of automated program repair approaches [7–9, 12, 13, 16–19, 25, 26, 28, 43, 44] have been proposed, and many of them fall into the category of test-based program repair. In test-based program repair, the repair tool takes a faulty program and a test suite including at least one failing test that reveals the fault as input and then generates a patch that makes all tests pass. However, test suites in real world projects are often weak [34], and a patched program passing all the tests may still be faulty. We call a patch *plausible* if the patched version passes all tests in the test suite, and we consider a patch *correct* if it fixes and only fixes the bug. As studied by Long et al. [20], the test suites in real world systems are usually weak such that most of the plausible patches are incorrect, making it difficult for a test-based program repair system to ensure the correctness of the patches. As existing studies [24, 34, 37] show, multiple automatic program repair systems produce much more incorrect patches than correct patches on real world defects, leading to low *precision* in their generated patches.

The low precision of existing program repair systems significantly affects the usability of these systems. Since test suites cannot guarantee the correctness of the patches, developers have to manually verify patches. When the precision of a program repair system is low, the developer has to verify a lot of incorrect patches, and it is not clear whether such a verification process is more costly than directly repairing the defect by the developers. An existing study [39] also shows that, when developers are provided with low-quality patches, their performance will drop compared to the situation where no patch is provided. As a result, we believe it is critical to improve the precision of program repair systems, even at the risk of losing some correct patches.

Since a weak test suite is not enough to filter out the incorrect patches produced by program repair systems, a direct idea is to enhance the test suite. Indeed, existing studies [42, 46] have attempted to generate new test cases to identify incorrect patches. However, while test inputs can be generated, test oracles cannot be automatically generated in general, known as the oracle problem [1, 31]. As a result, existing approaches either require human to determine test results [42], which is too expensive in many scenarios, or rely on inherent oracles such as crash-free [46], which can only identify certain types of incorrect patches that violate such oracles.

Our goal is to classify patches heuristically without knowing the full oracle. Given a set of plausible patches, we try to determine whether each patch is likely to be correct or incorrect, and reject the patches that are likely to be incorrect. Our approach is based on two key observations.

- **PATCH-SIM**. After a correct patch is applied, a passing test usually behaves similarly as before, while a failing test usually behaves differently.
- **TEST-SIM**. When two tests have similar executions, they are likely to have the same test results, i.e., both triggering the same fault or both are normal executions.

PATCH-SIM allows us to test patches without oracles, i.e., we run the tests before and after patching the system and check the degree of behavior change. As our evaluation will show later, PATCH-SIM alone already identify a large set of incorrect patches. However,

The authors acknowledge the anonymous reviewers for the constructive comments and revision suggestions. This work is supported by the National Key Research and Development Program under Grant No. 2016YFB1000105, and National Natural Science Foundation of China under Grant No. 61725201, 61529201, 61725201, 61672045. Lu Zhang is the corresponding author. Xinyuan Liu and Muhan Zeng are equal contributors to the paper and their names are sorted alphabetically.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180182

we can only utilize the original tests but not the newly generated test inputs as we do not know whether they pass or fail. TEST-SIM complements PATCH-SIM by determining the test results of newly generated test inputs.

Based on these two key observations, our approach consists of the following steps. First, we generate a set of new test inputs. Second, we classify the newly generated test inputs as passing or failing tests by comparing them with existing test inputs. Third, we determine the correctness of the patch by comparing the executions before and after the patch for each test, including both the original and the generated tests.

We have realized our approach by designing concrete formulas to compare executions, and evaluated our approach on a dataset of 139 patches generated from previous program repair systems including jGenProg [24], Nopol [24], jKali [24], HDRepair [13], and ACS [43]. Our approach successfully filtered out 56.3% of the incorrect patches without losing any of the correct patches. The results indicate that our approach increases the precision of program repair approaches with limited negative impact on the recall.

In summary, the paper makes the following main contributions.

- We propose two heuristics, PATCH-SIM and TEST-SIM, which provide indicators for patch correctness.
- We design a concrete approach that automatically classifies patches based on the two heuristics.
- We have evaluated the approach on a large set of patches, and the results indicate the usefulness of our approach.

The rest of the paper is organized as follows. Section 2 first discusses related work. Section 3 motivates our approach with examples, and Section 4 introduces our approach in details. Section 5 introduces our implementation. Section 6 describes our evaluation on the dataset of 139 patches. Section 7 discusses the threats to validity. Finally, Section 8 concludes the paper.

2 RELATED WORK

Test-based Program Repair. Test-based program repair is often treated as a search problem by defining a search space of patches, usually through a set of predefined repair templates, where the goal is to locate correct patches in the search space. Typical ways to locate a patch include the follows.

- *Search Algorithms.* Some approaches use meta-heuristic [16, 18] or random [33] search to locate a patch.
- *Statistics.* Some approaches build a statistical model to select the patches that are likely to fix the defects based on various information sources, such as existing patches [12, 13, 19] and existing source code [43].
- *Constraint Solving.* Some approaches [2, 14, 25, 26, 28, 36] convert the search problem to a satisfiability or optimization problem and use constraint solvers to locate a patch.

While the concrete methods for generating patches are different, weak test suites problem still remains as a challenge to test-based program repair and may lead to incorrect patches generated. As our evaluation has shown, our approach can effectively augment these existing approaches to raise their precisions.

Patch Classification. Facing the challenge of weak test suites, several researchers also propose approaches for determining the correctness of patches. Some researchers seek for deterministic approaches. Xin and Reiss [42] assume the existence of a perfect oracle (usually manual) to classify test results and generate new test inputs to identify oracle violations. Yang et al. [46] generate test inputs and monitor the violation of inherent oracles, including crashes and memory-safety problems. Compared with them, our approach does not need a perfect oracle and can potentially identify incorrect patches that do not violate inherent oracles, but has the risk of misclassifying correct patches.

Other approaches also use heuristic means to classify patches. Tan et al. [38] propose anti-patterns to capture typical incorrect patches that fall into specific static structures. Our approach mainly relies on dynamic information, and as the evaluation will show, the two approaches can potentially be combined. Yu et al. [47] study the approach that filters patches by minimizing the behavioral impact on the generated tests, and find that this approach cannot increase the precision of existing program repair approaches. Compared with their approach, our approach classifies the generated tests and puts different behavioral requirements on different classes. Finally, Weimer et al. [40] highlight possible directions in identifying the correctness of patches.

Patch Ranking. Many repair approaches use an internal ranking component that ranks the patches by their probability of being correct. Patch ranking is a very related but different problem from patch classification. On the one hand, we can convert a patch classification problem into a patch ranking problem by setting a proper threshold to distinguish correct and incorrect patches. On the other hand, a perfect patch ranking method does not necessarily lead to a perfect patch classification method, as the threshold can be different from defect to defect.

There are three main categories of patch ranking techniques. The first ranks patches by the the number of passing tests. However, this category cannot rank plausible patches. The second category uses syntactic [2, 14, 25] and semantic distances [2, 14] from the original program to rank patches. As our evaluation will show later, our approach could significantly outperform both types of distances. The third category [13, 19, 43] learns a probabilistic model from existing rules to rank the patches. Our approach could complement these approaches: as our evaluation will show later, our approach is able to identify 50% of the incorrect patches generated by ACS, the newest approach in this category.

Approaches to the Oracle Problem. The lack of test oracle is a long-standing problem in software testing, and the summaries of the studies on this problem can be found in existing surveys [1, 31]. Among them, a few studies focus on automatically generating heuristic test oracles. For example, invariant mining could potentially mine invariants [4, 5] from passing test executions to classify new test inputs. However, the effect of such an application on patch correctness identification is still unknown as far as we are aware and remains as future work.

Other related work. Marinescu and Cadar [22] propose KATCH for generating tests to cover patches. Our approach could be potentially combined with KATCH to improve the quality of the generated tests. This is a future direction to be explored.

Mutation-based fault localization such Metallaxis [30] and MUSE [27] shares a similar observation to PATCH-SIM: when mutating a faulty location, passing tests would exhibit significantly smaller behavior

```
  public void draw(...) {
+   if (true) return ;
    ...
```

(a) An incorrect patch produced by jKali [34]

```
public void testDrawWithNullDataset() { ...
  JFreeChart chart = ChartFactory.
    createPieChart3D("Test", null,...);
  try {...
    chart.draw(...);
    success = true; }
  catch (Exception e) {
    success = false; }
  assertTrue(success);
}
```

(b) A failing test checking for a null dataset

```
public void testNullValueInDataset() { ...
  dataset.setValue(..., null);
  JFreeChart chart = createPieChart3D(dataset);
  try {...
    chart.draw(...);
    success = true; }
  catch (Exception e) {
    success = false; }
  assertTrue(success);
}
```

(c) A passing test checking for a null value in a dataset

Figure 1: An Incorrect Patch for Chart-15

change than failing tests. This duality between approaches in different domains indicates that the observation is general and could potentially be applied in more domains in future.

3 PATCH CORRECTNESS AND BEHAVIOR SIMILARITY

In this section, we analyze the relation between patch correctness and behavior similarity to motivate our approach. We define a patch as a pair of program versions, the original buggy version and the patched version. To simplify discussion, we assume the program contains only one fault. As a result, a failing test execution must trigger the fault and produce an incorrect output.

Weak Oracles and PATCH-SIM. As mentioned, a test suite may be weak in either inputs or oracles, or both, to miss such an incorrect patch. To see how weak oracles could miss an incorrect patch, let us consider the example in Figure 1. Figure 1(a) shows an incorrect patch generated by jKali [34] for defect Chart-15 in the defect benchmark Defects4J [11]. In this example, calling `draw` will result in an undesired exception if the receiver object is initialized with a null dataset. Figure 1(b) shows a test case that detects this defect by creating such an object and checking for exceptions. Figure 1(c) shows a passing test checking that drawing with a null value in the dataset would not result in an exception. This patch simply skips the whole method that may throw the exception. In both the passing test and the failing test, the oracle only checks that no exception is thrown, but does not check whether the output of the program is correct. As a result, since the patch prevents the exception, both tests pass.

As mentioned in the introduction, we rely on observation PATCH-SIM to validate patches. Given a patch, we would expect that the original program and the patched program behave similarly on a passing test execution, while behaving differently on a failing test execution. In this example, the passing test would draw something on the chart in the original program, but would skip `draw` method completely in the patched program, leading to significant behavioral difference. Based on this difference, we can determine the patch as incorrect.

```
   ...
+ if(repeat)
  for (int i = 0; i < searchList.length; i++) {
    int greater = replacementList[i].length()
      - searchList[i].length();
    if (greater > 0) increase += 3 * greater;
  }  ...
```

(a) An incorrect patch generated by Nopol [45]

```
  ...
  for (int i = 0; i < searchList.length; i++) {
+   if (searchList[i] == null ||
+       replacementList[i] == null) {
+     continue;
+   }
    int greater = replacementList[i].length()
      - searchList[i].length();
    if (greater > 0) increase += 3 * greater;
  } ...
```

(b) The correct patch generated by human developers

Figure 2: An Incorrect Patch for Lang-39

Weak Inputs and TEST-SIM. To see how the weak test inputs could lead to the misses of incorrect patches, let us consider the example in Figure 2. This example demonstrates an incorrect patch for Lang-39 in Defects4J produced by Nopol [45]. The original program would throw an undesirable exception when an element in `replacementList` or `searchList` is `null`. To prevent such an exception, the correct way is to skip those elements as shown in Figure 2(b). However, the generated patch in Figure 2(a) blocks the whole loop based on the value of `repeat`, which is a parameter of the method. Interesting, all existing tests, either previously passed or failed, happen to produce the correct outputs in the patched program. This is because (1) the value of `repeat` happens to be `true` in all passing tests and be `false` in all failing tests, and (2) the condition `greater>0` is not satisfied by any element in `searchList` and `replacementList` in the failing tests. However, enhancing the test oracles by PATCH-SIM is not useful because the behavior on passing tests remains almost the same as the original program while the behavior on failing tests changes a lot.

To capture those incorrect patches missed by weak test inputs, we need new test inputs. To utilize PATCH-SIM with new test inputs, we need to know whether the outputs of the tests are correct or not. To deal with this problem, we utilize observation TEST-SIM. We assume that, when the execution of a new test input is similar to that of a failing test, the new test input is likely to lead to incorrect output. Similarly, when the execution of a new test input is similar to that of a passing test, the new test input is likely to lead to correct output. Based on this assumption, we can classify new test inputs by comparing its execution with those of the existing test inputs.

In the example in Figure 2, for any new test input triggering this bug, there will be an exception thrown in the middle of the loop, which is similar to the executions of failing tests. On the other

hand, for any test input that does not trigger this bug, the loop will finish normally, which is similar to the executions of existing passing tests.

Measuring Execution Similarity. An important problem in realizing the approach is how we measure the similarity of two test executions. In our approach, we measure the similarity of *complete-path spectrum* [10] between the two executions. A complete-path spectrum, or CPS in short, is the sequence of executed statement IDs during a program execution. Several existing studies [3, 10, 35] show that spectra are useful in distinguishing correct and failing test executions, and Harrold et al. [10] find that CPS is among the overall best performed spectra.

For example, let us consider the two examples in Figure 1 and Figure 2. In both examples, the defect will lead to an exception, which further leads to different spectra of passing and failing tests: the passing tests will execute until the end of the method, while the failing tests will stop in the middle. Furthermore, the failing test executions will become different after the system is patched: no exception will be thrown and the test executes to the end of the method.

Multiple Faults. In the case of multiple faults in a program, the two heuristics still apply. When there are multiple faults, the failing tests may only identify some of them. We simply treat the identified faults as one fault and the rest of them as correct program, and the above discussion still applies.

4 APPROACH

4.1 Overview

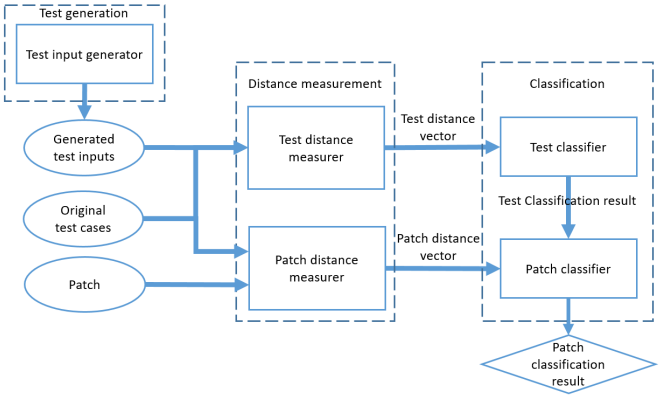


Figure 3: Approach overview

Figure 3 shows the overall process of our approach. We take the original buggy program, a set of test cases, and a patch as input, and produce a classification result that tells whether the patch is correct or not.

Our approach consists of five components classified into three categories: test generation (including *test input generator*), distance measurement (including *test distance measurer* and *patch distance measurer*) and result classification (including *test classifier* and *patch classifier*). First, *test input generator* generates a set of test inputs. We then run the generated tests on the original buggy program. During the test execution, we dynamically collect runtime information about the test execution. Based on the runtime information, *test distance measurer* calculates the distance between the executions of each newly generated test input and each original test case. A *distance* is a real number indicating how different two test executions are. The result is a vector of test distances. This vector is then passed to *test classifier*, which classifies the test as passing or failing by comparing its distances to passing tests and those to failing tests, based on TEST-SIM.

Now we have an enhanced set of test inputs which are classified as passing or failing and we can use them to determine patch correctness. Given a patch, *patch distance measurer* runs each test on the original program and the patched program and measure the distance between the two executions. The result is a vector of patch distances. Finally, this vector is taken into *patch classifier* which determines patch correctness by the distances, based on observation PATCH-SIM.

In the rest of this section, we introduce components in the three categories, respectively.

4.2 Test Generation

Given a program, *test generator* generates test inputs for this program. Furthermore, since our goal is to determine the correctness of the patch, we require the generated tests to cover the patched method. If a patch modifies multiple methods, the generated tests should cover at least one of them.

In theory, any test input generation techniques can be used in our approach. We can utilize symbolic execution techniques to cover the specific method, especially those designed for testing patches [22]. We can also adopt random testing techniques [6, 21, 29] and filter out those that do not cover any of the modified methods.

4.3 Distance Measurement

4.3.1 Measuring Distance. As mentioned previously, we measure the distance between two executions by comparing their complete-path spectra. As a result, the problem reduces to measuring the distances between two sequences. In general, there are many different metrics to measure sequence distances, such as longest common subsequence, Levenshtein distance, Hamming distance. As the first attempt in classifying patches based on sequence distances, we use the longest common subsequence (LCS) as a distance measure and leave other distance metrics to future work. An LCS of two sequences a and b is the longest sequence that can be obtained from both a and b by only deleting elements. We then normalize the length of LCS into a value between 0 and 1 using the following formula, where a and b are two sequences.

$$distance(a, b) = 1 - \frac{|LCS(a, b)|}{max(|a|, |b|)}$$

4.3.2 Test Distance Measurer. Component *test distance measurer* takes a generated test and calculates its distance with each original test. The result is a vector of distances, where each element represents a distance between a generated test and an original test.

To focus on the fault, we only consider the executed statements within the calling context of the patched methods. That is, we locate pairs of positions on the runtime trace: (a) entering a patched method from a method call and (b) leaving the method from the

same call and keep only the statements between the positions. This step could help us filter noises: two tests may be different in statement executions outside the calling context of the patched methods, but such a difference is often not related to the fault. If an original test does not cover any patched method, we also exclude the test from distance measurement.

4.3.3 Patch Distance Measurer. Component *patch distance measurer* takes each test, either generated or original and calculates the distance between its executions on the original program and on the patched program. The result is a vector of distances, where each element represents the distance of a test.

Different from *test distance measurer*, here we consider the full sequence of executed statements. This is because the compared executions come from the same test and they are unlikely to be noises outside the patched method.

4.4 Classification

Based on the distances, we can classify the generated tests and the patches. We describe the two components one by one.

4.4.1 Test Classifier. The *test classifier* classifies the test result of a generated test as passing or failing. Some generated tests are difficult to precisely classify and we discard these tests. Let $result(t)$ denotes the test result of the original test t, i.e., either *passing*, *failing*, or *discarded*. Let $distance(t, t')$ denotes the distance between the executions of t and t' on the original program. Given a generated test t', we use the following formulas to determine its classification result. The formula assigns the result of the nearest-neighbor to the generated test.

$$classification(t') = \begin{cases} passing & A_p < A_f \\ failing & A_p > A_f \\ discarded & A_p = A_f \end{cases}$$

where

$$A_p = min(\{distance(t, t') \mid classification(t) = passing\})$$
$$A_f = min(\{distance(t, t') \mid classification(t) = failing\})$$

Note that the above formula can only be applied when there is at least a passing test. If there is no passing test, we compare the distances with all failing tests with a threshold K_t and deem the test as *passing* if the test execution is significantly different from all failing tests based on the assumption that the original program works normally on most of the inputs. Please notice that there is always at least one failing test which exposes the defect.

$$classification(t) = \begin{cases} passing & K_t \leq A_f \\ failing & K_t > A_f \end{cases}$$

where

$$A_f = min(\{distance(t, t') \mid classification(t') = failing\})$$

4.4.2 Patch classifier. The *patch classifier* classifies a patch as *correct* or *incorrect* based on the calculated distances. Let $distance_p(t)$ denotes the distance between the executions of test t before and after applying the patch p. We determine the correctness of a patch p using the following formula.

$$classification(p) = \begin{cases} incorrect & A_p \geq K_p \\ incorrect & A_p \geq A_f \\ correct & \text{otherwise} \end{cases}$$

where

$$A_p = max(\{distance_p(t) \mid classification(t) = passing\})$$
$$A_f = mean(\{distance_p(t) \mid classification(t) = failing\})$$

This formula checks the two conditions in observation PATCH-SIM. First, the passing test should behave similarly. To check this condition, we compare the maximum distance on the passing tests with a threshold K_p and determine the patch as incorrect if the behavior change is too large. Second, the failing test should behave differently. However, since different defects require different ways to fix, it is hard to set a fixed threshold. As a result, we check whether the average behavior change in failing tests is still larger than all the passing tests. If not, the patch is considered incorrect.

We use the maximum distance for passing tests while using the average distance for failing tests. An incorrect patch may affect only a few passing tests, and we use the maximum distance to focus on these tests. On the other hand, after patched, the behaviors of all failing tests should change, so we use the average distance.

Please note this formula requires that we have at least a passing test, either original or generated. If there is no passing test, we simply treat the patch as correct.

5 IMPLEMENTATION

We have implemented our approach as a patch classification tool on Java. Given a Java program with a test suite and a patch on the program, our tool classifies the patch as correct or not.

In our implementation, we chose Randoop [29], a random testing tool, as the test generation tool. Since our goal is to cover the patched methods, testing generation tools aiming to cover a specific location seem to be more suitable, such as the tools based on symbolic executions [32] or search-based testing [6]. However, we did not use such tools because they are designed to cover the program with fewer tests. For example, Evosuite [6] generates at most three test cases per each buggy program in our evaluation subjects. Such a small number of tests are not enough for statistical analysis.

6 EVALUATION

The implementation and the evaluation data are available online. [1]

6.1 Research Questions

- RQ1: To what extent are TEST-SIM and PATCH-SIM reliable?
- RQ2: How effective is our approach in identifying patch correctness?

[1] https://github.com/Ultimanecat/DefectRepairing

- RQ3: How is our approach compared with existing approaches, namely, anti-patterns, Opad, syntactic similarity and semantic similarity?
- RQ4: How does test generation affect the overall performance?
- RQ5: How do the parameters, K_t and K_p, affect the overall performance?
- RQ6: What are the causes of false positive and false negatives?
- RQ7: How effective is our tool in classifying developers' correct patches?

RQ1 examines how much TEST-SIM and PATCH-SIM hold in general. RQ2 focuses on the overall effectiveness of our approach. In particular, we are concerned about how many incorrect and correct patches we filtered out. RQ3 compares our approach with four existing approaches for identifying patch correctness. Anti-patterns [38] capture incorrect patches by matching them with pre-defined patterns. Opad [46] is based on inherent oracles that patches should not introduce new crashes or memory-safety problem. Syntactic similarity [2, 14, 25] and semantic similarity [2, 14] are patch ranking techniques that rank patches by measuring, syntactically or semantically, how much their changes the program, which could be adapted to determine patch correctness by setting a proper threshold. RQ4 and RQ5 explore how different configurations of our approach could affect the overall performance. RQ6 investigates the causes of wrong results in order to guide future research. Finally, as will be seen in the next subsection, though we have tried out best to collect the generated patches on Java, the correct patches were still small in number compared with incorrect patches. To offset this, RQ7 further investigates the performance of our approach on the developers' correct patches.

6.2 Dataset

We have collected a dataset of generated patches from existing papers. Table 1 shows the statistics of the dataset. Our dataset consists of patches generated by six program repair tools. Among the tools, jGenProg is a reimplementation of GenProg [15, 16, 41] on Java, a repair tool based on genetic algorithm; jKali is a reimplementation of Kali [34] on Java, a repair tool that only deletes functionalities; Nopol [45] is a tool that relies on constraint solving to fix incorrect conditions and two versions, 2015 [23] and 2017 [45], are used in our experiment; HDRepair [13] uses information from historical bug fixes to guide the search process; ACS [43] is a tool based on multiple information sources to statistically and heuristically fix incorrect conditions. The selected tools cover the three types of patch generation approaches: search algorithms (jGenProg, jKali), constraint-solving (Nopol) and statistical (HDRepair, ACS). More details of the three types can be found in the related work section.

The patches generated by jGenProg, jKali and Nopol2015 are collected from Martinez et al.'s experiments on Defects4J [23]. The patches generated by Nopol2017 are collected from a recent report on Nopol [45]. Patches generated by HDRepair is obtained from Xin and Reiss' experiment on patch classification [42]. The patches generated by ACS is collected from ACS evaluation [43].

All the patches are generated for defects in Defects4J [11], a widely-used benchmark of real defects on Java. Defects4J consists of six projects: Chart is a library for displaying charts; Math is a library for scientific computation; Time is a library for date/time processing; Lang is a set of extra methods for manipulating JDK classes; Closure is optimized compiler for Javascript; Mockito is a mocking framework for unit tests.

Some of the patches are not supported by our implementation, mainly because Randoop cannot generate any tests for these patches. In particular, Randoop cannot generate any tests for Closure and Mockito. We removed these unsupported patches.

The patches from Martinez et al.'s experiments, the ACS evaluation and Qi et al.'s experiments contains labels identifying the correctness of the patches, which mark the patches as *correct*, *incorrect*, or *unknown*. The patches of Nopol2017 do not contain such labels. We manually checked whether the unlabeled patches and some labeled patches are semantically equivalent to the human-written patches. Since the patches whose correctness is unknown cannot be used to evaluate our approach, we remove these patches.

In the end, we have a dataset of 139 patches generated by automatic program repair tools, where 110 are incorrect patches and 29 are correct patches.

To answer RQ6, we also added all developer patches on Defects4J into our dataset. Same as generated patches, we removed the unsupported patches, including all patches on Closure and Mockito. In the end, we have 194 developer patches. Please note that developer patches are only used in RQ6 since they have different characteristics compared with generated patches.

6.3 Experiment Setup

Test Generation. We kept Randoop to run 3 minutes on the original program and collected the tests that covered the patched methods. We stop at 3 minutes because for most defects, Randoop produced enough tests within three minutes, and for the remaining defects that do not have enough tests, lengthening the time would not lead to more tests. We then randomly selected 20 tests for each patch. If there were fewer than 20 tests, we selected all of them. In the end, we have 7.1 tests per patch in average, with a minimum of 0 test. Based on the classification of TEST-SIM, 71% of the generated tests are passing tests.

RQ1. To evaluate PATCH-SIM, we measured the average distance between test executions of patched and unpatched versions, and check whether there is significant differences between passing and failing tests on correct and incorrect patches. To evaluate TEST-SIM, we measured the distances between tests, and analyzed whether closer distances indicate similar test results.

RQ2. We applied our approach to the patches in our dataset and checked whether our classification results are consistent with the labels about correctness.

RQ3. We applied the four existing approaches to the dataset and compared their results with our result.

Anti-patterns was originally implemented in C. To apply anti-patterns on Java dataset, we took the seven anti-patterns defined by Tan et al. [38] and manually checked whether the patches in our dataset fall into these patterns.

Opad [46] uses inherent oracles that patches should not introduce new crash or memory-safety problems. Opad was originally designed for C and we need to adapt it for Java. On the one hand,

Table 1: Dataset

Project	jGenprog			jKali			Nopol2015			Nopol2017			ACS			HDRepair			Total(Generated)			Developer Patches		
	P	C	I	P	C	I	P	C	I	P	C	I	P	C	I	P	C	I	P	C	I	P	C	I
Chart	6	0	6	6	0	6	6	1	5	6	0	6	2	2	0	0	0	0	26	3	23	25	25	0
Lang	0	0	0	0	0	0	7	3	4	4	0	4	3	1	2	1	0	1	15	4	11	58	58	0
Math	14	5	9	10	1	9	15	1	14	22	0	22	15	11	4	7	2	5	83	20	63	84	84	0
Time	2	0	2	2	0	2	1	0	1	8	0	8	1	1	0	1	1	0	15	2	13	27	27	0
Total	22	5	17	18	1	17	29	5	24	40	0	40	21	15	6	9	3	6	139	29	110	194	194	0

P=Patches, C=Correct Patches, I=Incorrect Patches

crashes are represented as runtime exceptions in Java. On the other hand, memory-safety problems are either prevented by the Java infrastructure or detected as runtime exceptions. Therefore, we uniformly detect whether a patch introduces any new runtime exception on test runs. If so, the patch is considered incorrect.

Regarding syntactic and semantic distances, different papers [2, 14, 25] have proposed different metrics to measure the syntactic and semantic distances and a summary can be found in Table 2. However, as analyzed in the table, many of the metrics are defined for a specific category of patches and cannot be applied to general patches. In particular, many metrics are designed for expression replacement only and their definitions on other types of changes are not clear. As a result, we chose the two metrics marked as "general" in Table 2 for comparison: one measuring syntactic distance by comparing AST and one measuring semantic distance by comparing complete-path spectrum.

The AST-based syntactic distance is defined as the minimal number of AST nodes that need to be deleted or inserted to change the one program into the other program. For example, changing expression $a > b + 1$ to $c < d + 1$ needs to at least remove three AST nodes (a, b, $>$) and insert three AST nodes (c, d, $<$), giving a syntactic distance of 6. The semantic distance based on complete-path spectrum for program p is defined using the following formula, where T_o is the set of all original tests that cover at least one modified method and $distance_p$ is defined in Section 4.4.2.

$$LED(p) = mean(\{distance_p(t) \mid t \in T_o\})$$

The syntactic/semantic distance gave a ranked list of the patches. Then we checked if we could find an optimal threshold to separate the list into correct and incorrect patches.

RQ4. We considered two different test generation strategies and compared their results with the result of RQ1.

- **No generation.** This strategy simply does not generate any test input. This strategy serves as a baseline for evaluating how much the newly generated test inputs contribute to the overall performance.
- **Repeated Randoop runs.** Since Randoop is a random test generation tool, different invocations to Randoop may lead to different test suites. This strategy simply re-invokes Randoop to generate a potentially different set of tests and the comparison helps us understand the effect of the randomness in test generation on the overall performance of our approach.

Since the second strategy involves re-generating the tests and is expensive to perform, we evaluated this strategy on a randomly selected subset of 50 patches. To see the effect of randomness, we repeated the experiments for 5 times.

RQ5. During the experiments for the first three research questions, we set the parameters of our approach as follows: $K_p = 0.25$, $K_t = 0.4$. These parameter values are determined by a few attempts on a small set of patches.

To answer RQ4, we systematically set different values for parameters K_p and K_t and then analyzed how these parameters affect our result on the whole dataset.

RQ6. We manually analyzed all false positives and false negatives to understand the causes of false classification and summarize the reasons.

RQ7. We applied our approach on human-written patches provided by Defects4J benchmark and check whether our approach misclassified them as incorrect or not.

Hardware Platform. The experiment is performed on a server with Intel Xeon E3 CPU and 32GB memory.

6.4 Result of RQ1: Reliability of Heuristics

Table 3 shows the results of evaluating PATCH-SIM, i.e., the distances between test executions on patched and unpatched versions. As we can see from the table, for correct patches, the distances of passing tests are very close to zero, while failing tests have a much larger distances that is 9.5 times of passing tests. The result indicates that PATCH-SIM holds in general. On the other hand, the passing tests and the failing tests do not exhibit such a strong property on incorrect patches. While the distances of failing tests are still larger than passing tests, the ratio is only 1.32 times rather than 9.5 times. This results indicate that PATCH-SIM can be used to distinguish correct and incorrect patches.

Figure 4 shows the results of evaluating TEST-SIM. The X-axis shows intervals of distances while the Y-axis shows the percentage of tests fall into the intervals. As we can see from the figure, when two tests have a short distance, they are more likely to have the same test results rather than different test results. This result indicates that TEST-SIM holds in general. On the other hand, when the two tests have a long distance, they are more likely to have different test results rather than the same test results.

6.5 Result of RQ2: Overall Effectiveness

Table 4 and Table 5 shows the performance of our approach on the dataset per tool and per project, respectively. As shown in the tables, our approach successfully filtered out 62 of 110 incorrect plausible patches and filtered out no correct patch. Furthermore, our

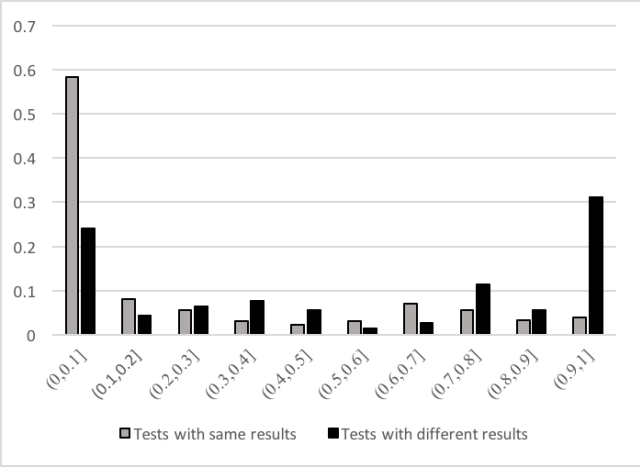
Table 2: Different syntactic and semantic metrics

Metric	Type	Scope	Description
AST-based [14, 25]	Syn	General	Number of AST node changes introduced by the patch.
Cosine similarity [14]	Syn	Replacement	The cosine similarity between the vectors representing AST node occurrences before and after the change. It is not clear how to apply it to insertions and deletions because there will be a zero vector and cosine similarity cannot be calculated.
Locality of variables and constants [14]	Syn	Expression Replacement	The distance is measured by the Hamming distance between the vectors representing locations of variables and constants. It is not clear how to apply it to patches with multiple changes.
Expression size distance [2]	Syn	Expression Replacement	The distance is 0 when two expressions are identical, otherwise the size of the affected expression after patching. It is not clear how to apply it to changes other than expression replacement.
Complete-path spectrum [2]	Sem	General	The difference between complete-path spectra.
Model counting [14]	Sem	Boolean Expression replacement	The distance is measured by the number of models that make two expressions evaluate differently. The definition is bounded to Boolean expressions.
Output coverage [14]	Sem	Programs with simple outputs	The distance is measured by the proportion of different outputs covered by the patched program. It is not clear how to define "output" in general for complex programs.

"Syn" and "Sem" stand for syntactical distance and semantic distance respectively.

Table 3: PATCH-SIM

	Passing Tests	Failing Tests
Incorrect Patches	0.25	0.33
Correct Patches	0.02	0.19



X-axis: intervals of distance on tests Y-axis: percent of tests

Figure 4: TEST-SIM

Table 4: Overall Effectiveness per Tool

Tool	Incorrect	Correct	Incorrect Excluded	Correct Excluded
jGenprog	17	5	8(47.1%)	0
jKali	17	1	9(52.9%)	0
Nopol2015	24	5	16(66.7%)	0
Nopol2017	40	0	22(55.0%)	0
ACS	6	15	3(50.0%)	0
HDRepair	6	3	4(66.7%)	0
Total	110	29	62(56.3%)	0

"In/correct Excluded" shows the number of patches that are filtered out by our approach and are in/correct.

Table 5: Overall Effectiveness per Project

Project	Incorrect	Correct	Incorrect Excluded	Correct Excluded
Chart	23	3	14(60.9%)	0
Lang	11	4	6(54.5%)	0
Math	63	20	33(52.4%)	0
Time	13	2	9(69.2%)	0
Total	110	29	62(56.3%)	0

approach shows similar performance on different tools and different projects, indicating that our results are potentially generalizable to different types of projects and different types of tools.

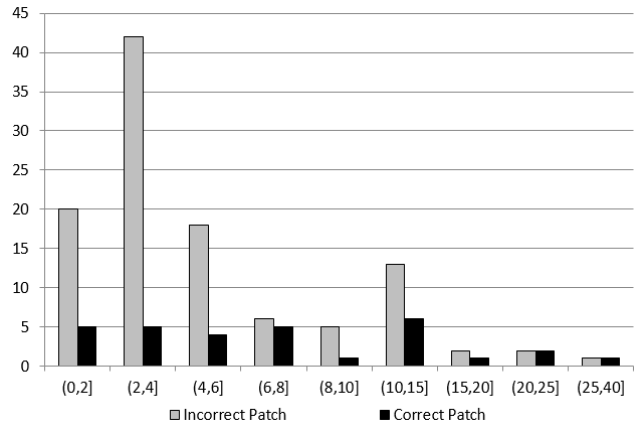
Please note that although our approach did not filter out any correct patch on our dataset, in theory it is still possible to filter out correct patches. For example, a patch may significantly change the control flow of a passing test execution, e.g., by using a new algorithm or calling a set of different APIs, but the test execution could produce the same result. However, given the status of current program repair approaches, such patches are probably scarce. When applying our approach on human-written patches, some correct patches are filtered out. More details for the effectiveness on human-written patch is discussed in RQ6.

Our approach took about 5 to 10 minutes to determine the correctness of a patch in most cases, while some patches might take up to 30 minutes. Most of the time was spent on generating the test inputs and recording the runtime trace.

6.6 Result of RQ3: Comparing with Others

Anti-patterns. Among all 139 patches, anti-patterns filtered out 28 patches, where 27 are incorrect and 1 is correct. The result shows that our approach significantly outperforms anti-patterns. Furthermore, 13 of the 27 incorrect patched filtered out by anti-patterns were also filtered out by our approach, while the remaining 14 patches were not filtered out by our approach. This result suggests that we may potentially combine the two approaches to achieve a better performance.

Opad. When applied with the same set of test inputs as our approach, Opad failed to recognize any of the incorrect patches. To further understand whether a stronger test suite could achieve better results, we further selected up to 50 tests instead of 20 tests for each patch. This time Opad filtered out 3 incorrect patches. This result suggests that inherent oracles may have a limited effect on classifying Java patches, as the Java infrastructure has already prevented a lot of crashes and memory safety problems and it may not be very easy for a patch to break such an oracle.

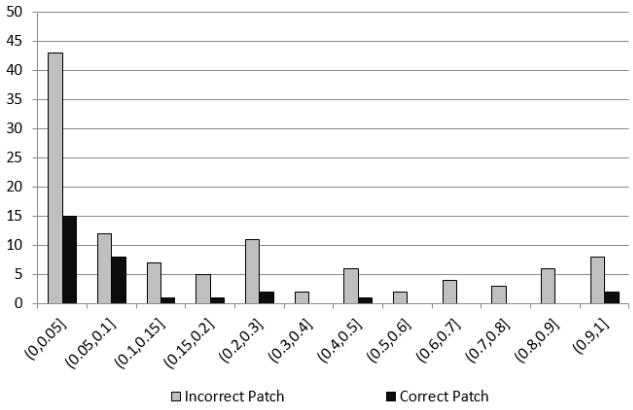


X-axis: intervals of syntactic distance Y-axis: numbers of patches

Figure 5: Syntactic distance

Syntactic and Semantic Distance. Fig. 5 shows the distribution of incorrect patches and correct patches on syntactic distance. The x-axis shows the intervals of distances while Y-axis shows the numbers of patches within the intervals. As we can see from the figure, the incorrect patches and correct patches appear in all intervals and the distribution shows no particular characteristics. If we would like to exclude 56.3% incorrect patches using syntactic distance, we need to at least exclude 66.7% of the correct patches. This result indicates that syntactic distance cannot be directly adapted to determine patch correctness.

Fig. 6 shows the distribution of incorrect and correct patches on semantic distance. As we can see from the figure, both types of patches tend to appear more frequently when the distance is small.



X-axis: intervals of semantic distance Y-axis: numbers of patches

Figure 6: Semantic distance

When the distance grows larger, both types decrease but correct patches decrease faster. If we would like to exclude 56.3% incorrect patches using semantic distance, we need to exclude 43.3% correct patches. This result indicates that semantic distance could be a better measurement than syntactic distance in determining patch correctness, but is still significantly outperformed by our approach.

Please note that the above results do not imply that syntactic and semantic distances are not good at ranking patches. While it is difficult to find a threshold to distinguish correct and incorrect patches for a group of defects, it is still possible that the correct patches are ranked higher on most individual defects.

6.7 Result of RQ4: Effects of Test Generation

Table 6 shows the result without generating test inputs. Without the generated test inputs, our approach filtered out 8 less incorrect patches and still filtered 0 correct patch. This result suggests that PATCH-SIM alone already makes an effective approach, but test generation and TEST-SIM can further boost the performance non-trivially.

Table 6: Comparison with no test generation

	Default Approach	No Generation
Incorrect Excluded	62	54
Correct Excluded	0	0

Regarding randomness, we repeated our experiment on the selected 50 patches 5 times and got different results on only 3 incorrect patches. The best case had only one more excluded incorrect patch than the worst case. The result shows that randomness does affect the result, but the impact is limited.

6.8 Result of RQ5: Parameters

Two parameters are involved in our approach, K_t and K_p, both ranging from 0 to 1. The results of our approach with different parameters are shown in Table 7 and 8, where each column shows the result with the parameter value in the table head. As we can see

Table 7: Parameter K_p

	0.05	0.1	0.15	0.25	0.3	0.4	0.5	0.6	0.8	1
IE	71	66	62	62	60	57	56	56	55	54
CE	4	1	0	0	0	0	0	0	0	0

IE = Incorrect Excluded, CE = Correct Excluded

Table 8: Parameter K_t

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
IE	65	62	62	62	62	60	60	60	60	59	59
CE	4	1	0	0	0	0	0	0	0	0	0

IE = Incorrect Excluded, CE = Correct Excluded

from the tables, setting the parameters to different values have a limited impact on the overall results and a large range of parameter value could achieve the best performance. The result indicates that our approach does not require a precise tuning of parameters.

6.9 Result of RQ6: Causes of Wrong Result

Our approach gave wrong results on 47 incorrect patches. We manually analyzed these patches and identified three main causes of the wrong classification, as follows.

Too weak test suite. It is often the case (21 out of 48) that only one failing test covers the patched method. Without passing test, our approach only relies on the threshold K_t to classify the tests and it is sometimes difficult to generate tests that pass the threshold. As a result, we might have no or only a few passing tests to perform the patch classification, leading to a low performance.

Unsatisfying test generation. Another common case (27 out of 48, 9 overlap with the previous case) is that test generation tool fails to generate satisfying tests. Randoop might fail to give tests that cover the patched method or fail to generate tests that could expose the incorrect behavior. The patches in this category have the potential to be correctly identified if we use a stronger test generation tool.

Unsatisfying classification formula. The final case (8 out of 48) was caused by large behavior changes in some failing test execution. Since we calculated the average distance of all failing test executions in the patch classification, if there was a very large value, the average value might become large even if all the rest failing tests had small behavior changes. As a result, the patch may be misclassified. This problem may be fixed by generating more failing test cases to lower down the average value, or to find a better formula to classify the patches.

6.10 Result of RQ7: Developer Patches

Among the 194 correct developer patches, our tool classified 16 (8.25%) patches as incorrect. We further analyzed why the 16 patches are misclassified and found that all the 16 patches have non-trivially changed the control flow and caused a significant difference in CPS in the passing test executions. In particular, the behaviors of passing tests have significantly changed in 6 patches, while in the rest 10 patches the behaviors remain almost the same but the executed statements changed significantly (e.g., calling a different method with the same functionality). The results imply that (1) human patches are indeed more complex than those generated by current automated techniques; (2) when the complexity of patches grows, our approach is probably still effective as only a small portion of correct patches is excluded; (3) To further enhance the performance, we need to enhance PATCH-SIM and CPS to deal with such situations.

7 THREATS TO VALIDITY AND LIMITATIONS

The main threat to internal validity is that we discarded some patches from our dataset, either because their correctness cannot be determined, or because the infrastructure tool used in our implementation cannot support these patches. As a result, a selection bias may be introduced. However, we believe this threat is not serious because the removed patches are small in number compared with the whole dataset and the results on these patches are unlikely to significantly change the overall results.

The main threat to external validity is whether our approach can be generalized to different types of program repair tools. While we have selected repair tools from all main categories of program repair tools, including tools based on search algorithms, constraint solving and statistics, it is still unknown whether future tools will have characteristics significantly different from current tools. To minimize such a threat, we have added RQ7 to test on developer patches, which can be viewed as the ultimate goal of automatically generated patches. The results indicates that our approach may have different performance on developer patches and generated patches, but the difference is limited.

The main threat to construct validity is that the correctness of the patches are manually evaluated and the classification may be wrong. To reduce this threat, all difficult patches are discussed through the first two authors to make a mutual decision. Furthermore, part of the classification comes from Martinez et al.'s experiment [23], whose results have been published online for a few years and there is no report questioning the classification quality as far as we are aware.

There can be many different choices in designing the formulas. For example, we can use a different sequence distance or even a different spectrum to measure the distance of two executions. We can use different statistical methods for classifying tests and patches. The current paper does not and cannot explore all possibilities and leave them as future work.

8 CONCLUSION

In this paper, we have proposed an approach to automatically determining the correctness of the patches based on behavior similarities between program executions. As our evaluation shows, our approach could effectively filter out 56.3% of the incorrect patches generated without losing any of the correct patches. The result suggests that measuring behavior similarity can be a promising way to tackle the oracle problem and calls for more research on this topic.

REFERENCES

[1] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[2] Loris DâĂŹAntoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification*. Springer, 383–401.

[3] William Dickinson, David Leon, and Andy Podgurski. 2001. Pursuing failure: the distribution of program failures in a profile space. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*, A. Min Tjoa and Volker Gruhn (Eds.). ACM, 246–255. https://doi.org/10.1145/503209.503243

[4] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Trans. Software Eng.* 27, 2 (2001), 99–123. https://doi.org/10.1109/32.908957

[5] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1-3 (2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[6] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *ESEC/FSE*. ACM, 416–419.

[7] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-Leak Fixing for C Programs. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 459–470.

[8] Qing Gao, Hansheng Zhang, Jie Wang, and Yingfei Xiong. 2015. Fixing Recurring Crash Bugs via Analyzing Q&A Sites. In *ASE*. 307–318.

[9] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *AAAI*. 1345–1351.

[10] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. 1998. An Empirical Investigation of Program Spectra. In *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE '98, Montreal, Canada, June 16, 1998*, Thomas Ball, Frank Tip, and A. Michael Berman (Eds.). ACM, 83–90. https://doi.org/10.1145/277631.277647

[11] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*. 437–440.

[12] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *ICSE '13*. 802–811.

[13] Xuan-Bach D Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 1. IEEE, 213–224.

[14] 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 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 593–604. https://doi.org/10.1145/3106237.3106309

[15] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *ICSE*. 3–13.

[16] C. Le Goues, ThanhVu Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *Software Engineering, IEEE Transactions on* 38, 1 (Jan 2012), 54–72.

[17] Xuliang Liu and Hao Zhong. 2018. Mining StackOverflow for Program Repair. (2018), to appear pages.

[18] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 166–178. https://doi.org/10.1145/2786805.2786811

[19] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 298–312. https://doi.org/10.1145/2837614.2837617

[20] Fan Long and Martin C. 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, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 702–713. https://doi.org/10.1145/2884781.2884872

[21] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. 2015. GRT: Program-analysis-guided random testing. In *ASE*. 212–223.

[22] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-coverage Testing of Software Patches. In *ESEC/FSE*. 235–245.

[23] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2016. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering* (2016), 1–29.

[24] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA, Demonstration Track*. 441–444. https://doi.org/10.1145/2931037.2948705

[25] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *ICSE*. 448–458.

[26] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *ICSE*. 691–701.

[27] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *ICST*. 153–162.

[28] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *ICSE*. 772–781.

[29] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *OOPSLA*. 815–816.

[30] Mike Papadakis and Yves Le Traon. 2012. Using Mutants to Locate "Unknown" Faults. In *ICST*. 691–700.

[31] Mauro Pezze and Cheng Zhang. 2015. Automated Test Oracles: A Survey. *Advances in Computers* 95 (2015), 1–48.

[32] Corina S. Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *ASE*. 179–180.

[33] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *ICSE*. 254–265.

[34] Zichao Qi, Fan Long, Sara Achour, and Martin C. Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*. 24–36.

[35] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. 1997. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ESEC/FSE*. Springer, 432–449.

[36] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *PLDI*. 15–26.

[37] 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 *FSE*. 532–543.

[38] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in Search-Based Program Repair. In *FSE*. 727–738.

[39] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically Generated Patches As Debugging Aids: A Human Study. In *FSE*. 64–74.

[40] Westley Weimer, Stephanie Forrest, Miryung Kim, Claire Le Goues, and Patrick Hurley. 2016. Trusted Software Repair for System Resiliency. In *DSN-W*. 238–241.

[41] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *ICSE '09*. 364–374.

[42] Qi Xin and Steven Reiss. 2017. Identifying Test-Suite-Overfitted Patches through Test Case Generation. In *ISSTA*.

[43] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *ICSE*.

[44] Yingfei Xiong, Hansheng Zhang, Arnaud Hubaux, Steven She, Jie Wang, and Krzysztof Czarnecki. 2015. Range fixes: Interactive error resolution for software configuration. *Software Engineering, IEEE Transactions on* 41, 6 (2015), 603–619.

[45] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopoly: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* (2016).

[46] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better Test Cases for Better Automated Program Repair. In *FSE*.

[47] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2017. Test Case Generation for Program Repair: A Study of Feasibility and Effectiveness. *CoRR* abs/1703.00198 (2017).