PatchID: A Overfitting Patches Identification Method for Automated Program Repair

Xuan Zhou[1,2], Xingqi Wang[1,2*], Dan Wei[1,2] and Yanli Shao[1,2]

[1*]School of Computer Science, Hangzhou Dianzi University, Hangzhou, 310018, Zhejiang, China.

[2]Key Laboratory of Discrete Industrial Internet of Things of Zhejiang Province, Hangzhou, 310018, Zhejiang, China.

*Corresponding author(s). E-mail(s): xqwang@hdu.edu.cn;
Contributing authors: 212050276@hdu.edu.cn;
weiwd@hdu.edu.cn; shaoyanli@hdu.edu.cn;

Abstract

The patch generated by automatic program repair (APR) needs to be verified. APR usually uses the test suite as the criterion for verifying the correctness of the patch. However, the test suite is unable to fully represent the oracle of the program, which causes APR to generat a large number of overfitting patches that not only fail to fix the original error, but also cause new errors. In order to reduce the number of overfitting patches generated by APR, this paper proposes an overfitting patches identification method for APR, PatchID. The core idea of PatchID is that the dynamic behavior of the passing tests between the buggy program and the correct patch is the same, however the dynamic behavior of the failing tests between themis different. The algorithm first constructs the dynamic behavior expressions that cause the bug from the program and the test suite, then generates new tests to enhance the original test suite, and finally gets the same dynamic behavior expressions from the patch, and identifies whether the patch is overfitting according to whether the value of the dynamic behavior expression changes with the use of the patch. The paper is evaluated on two datasets consisting of 157 Defects4J patches and 380 Java+JML patches, respectively. PatchID successfully identifies 63 overfitting patches and 15 correct patches on the first dataset, and 169 overfitting patches on the second dataset. In addition, PatchID classifies overfitting patches into three types of patches. Experiments show that the method proposed in this paper is superior to the existing similar methods.

Keywords: automatic program repair, overfitting patch, program state abstract, test generation

1 Introduction

Automatic program repair (APR) has been widely studied in the past decade, and a large number of repair technologies have been proposed. Especially after the emergence of GenProg(Forrest et al, 2009), the APR based on test suite occupies the vast majority. The test suite based APR takes a given test suite as oracle and generates patches that are considered correct if they passes the test suite. Accurally, the test suite is weak and can not fully represent the oracle of the program, which results in the existence of patches that pass all the tests but are still wrong, that is, overfitting patches. As a result, APR technology produces a large number of invalid patches. Current APR technologies are far from mature, and most of them will simply accept patches that pass the test suite. According to Xin (2017a), the majority of patches generated by GenProg, AE and RSRepair are incorrect. More recent techniques try to find many other strategies (e.g., using human-written patches, repair templates and condition synthesis, bug-fixing instances and forbidden modifications) for repair. However their repair performance is still lower.

Due to the low performance of current APR techniques, software developers have to verify a large amount of patches manually to filter out overfitting patches, which consumes too many resources. Therefore, it has become an urgent problem to identify overfitting patches. If a patch passes the test suite and is considered the correct patch, then simply enhancing the test suite can reduce the number of overfitting patches. However, automatic test generation tools can only generate test inputs, and the appropriate test outputs still need to be determined manually. Even so, these approaches still fail to express a complete oracle. It is very difficult to have a complete oracle for a program, especially for large projects. Nilizadeh(2021) used a formal method (specification and verification), JML as correctness criterion to successfully describe a small project. However facing the large project, such as Defects4J(Just et al, 2014), JML fails. And it seems impossible to use JML to describe such a large project.

At present, it is already very helpful to identify whether a patch is overfitting or not because quickly identifying overfitting patches can improve the success rate of APR technology. Furthermore, if there is technology to subdivide overfitting patches, it can speed bugs fixing. According to Yu (2019), overfitting patches can be divided into the following three categories:

- A-Overfitting Patch: The patch does not completely fix the incorrect behavior nor does it destroy the original correct behavior.
- B-Overfitting Patch: The patch that fixes the original incorrect behavior but destroys the original correct behavior, which is also called regression.

- AB-Overfitting Patch: The patch destroys the original correct behavior instead of fixing the incorrect behavior.

Because the test suite is not complete, it is far from sufficient to use the consistency of the actual output of the program and the test output, that is to say, the current APR technology does not take full advantage of the test suite, only uses the input and the output of the test suite, and does not dig into the hidden information in the test suite. At present, different overfitting identification methods have been proposed. Among them, the strategy proposed by Xiong(2018) is the first technology to mine the deep behavior of test suites and programs. Through the ideas of TEST-SIM and PATCH-SIM, the identification rate of overfitting patch can reach 56%. The experiment of Yang(2020) shows that patches can modify the behavior of the program. They try to use program invariants to describe the program behavior and observe the program oracle from another view. Inspired by this, we propose a new technique, PatchID, to solve the problem of patch overfitting.

The goal of PatchID is to determine the category of a patch by digging into the correct behavior of the program based on a given test suite. While patches are able to pass the test suite, the passing tests reflect the correct behavior of the program, and the failing test reflect the wrong behavior of the program. From this point of view, patches should maintain the correct behavior of the program and modify the wrong behavior of the program. We believe that there are specific relationships between variables in a correct program, which are reflected by their values at runtime, and these relationships are a manifestation of the correct behavior of the program. So PatchID identifies a patch based on the following two important observations, which are from the perspective of program runtime variables:

- PATCH-SIM: After using the patch, the dynamic behavior of the program of the passing test (the specific relationship between variables, which is represented by a 5-tuple in this paper) is similar to that before, while the dynamic behavior of the program of the failing test is different.
- TEST-SIM: When two tests have the same dynamic behavior, the two tests belong to the same category, that is, they belong to either the passing test or the failing test.

Based on the above two points, this paper designs and implements PatchID, and verifies the method on a dataset consisting of 157 patches. These patches are generated by APR techniques based on Defects4J, including HDRepair, jGenprog(Martinez and Monperrus, 2016), ACS(Xiong et al, 2017), jKail(Qi et al, 2015), and Nopol(Xuan et al, 2016).The method presented in this paper successfully identified 63 overfitting patches and 15 correct patches from 157 patches. In addition, PatchID further subdivides the 63 overfitting patches into three categories, which is not available in other technologies at present. We also verified on the dataset composed of 380 Java+JML patches, and successfully identified 169 patches.To sum up, the contributions of this paper are:

- 5-tuple for computing program execution similarity is proposed.
- An overfitting patch identification and subdivision algorithm is designed for automatic patch generation.
- The framework of overfitting patch identification is implemented and the proposed method is evaluated. The results show that the proposed method is effective.

The rest of the paper is organized as follows. Section 2 begins with a discussion of related work. Section 3 describes some of the necessary concepts involved in the approach of this paper. Section 4 describes the method in this paper in detail. Section 5 presents an experimental evaluation of the method presented in this paper. Finally, the paper is summarized in Section 6.

2 Related work

Automatic Program Repair. Generally, automatic program repair includes three steps: fault location, patch generation and patch verification. They will first generate some candidate patches, then use the existing test suite to verify these candidate patches, and the candidate patches that pass the test suite will be regarded as the correct patches. At present, the existing methods can be roughly divided into the following three categories:

- Heuristic Search-Based APR: Heuristic search-based automatic repair technology generates the repair patch by artificially defined heuristic rules. This kind of algorithms inlcude GenProg, ARJA-e(Yuan and Banzhaf, 2018, 2020), PraPR(Kim et al, 2013), etc, which use genetic algorithm to generate patches with the original program as search space. In addition, there are approaches, such as HistoricalFix(Le et al, 2016), CapGen(Wen et al, 2018), ConFix(Kim and Kim, 2019) to generate patches from historical repair patches and those as SCRepair(Ji et al, 2016), CRSearcher(Wang et al, 2017), SSFix(Xin and Reiss, 2017b), SimFix(Jiang et al, 2018), Refactory(Hu et al, 2019) to generate patches bycode similarity.
- Template-Based APR: It generates patches based on manual template. According to the experience of developers or researchers, some patch templates or patch generation strategies are predefined to guide the repair process. PAR(Ghanbari et al, 2019), iFixR(Koyuncu et al, 2019), SapFix(Marginean et al, 2019), ErrDoc(Tian and Ray, 2017), BovInspector(Gao et al, 2016), LeakFix(Gao et al, 2015), AutoFix(Yan et al, 2016), GumTree(Falleri et al, 2014), SketchFix(Hua et al, 2018), NPEfix(Durieux et al, 2017), F1X(Mechtaev et al, 2018a), HERCULES(Saha et al, 2019), and the anti-pattern method proposed by Tan(Tan et al, 2016) fall into this category of algorithms.
- Semantic Constraint-Based APR: Semantic Constraint-Based APR infers the correct specification of the program by some means as a constraint to guide the patch generation or to verify the correctness of the patch. SemFixNguyen et al (2013), DirectFix(Mechtaev et al, 2015), Angelix(Mechtaev

et al, 2016), AllRepair(Rothenberg and Grumberg, 2016), Nopol(Xuan et al, 2016), S3(Le et al, 2017), SemGraft(Mechtaev et al, 2018b), MemFix(Lee et al, 2018), FootPatch(van Tonder and Goues, 2018), SearchRepair(Ke et al, 2015), SOSRepair(Afzal et al, 2019) belong to this kind of algorithms.

All these repair techniques use G & V method to produce patches. For these technologies, if a patch passes the test suite, it is the correct patch. However, a program cannot be fully tested, and the test suites given by existing datasets are all weak(Yang et al, 2017), so repair techniques will always produce a large number of overfitting patches on these datasets. New techniques are needed to find to identify these overfitting patches.

Patch identification. Facing the problem of overfitting patches, researchers have proposed many methods. Xin(2017a) identifys patches by generating new test inputs and checking semantic differences between buggy programs and patches. Nilizadeh(2021) uses JML as an oracle on small programs to verify the effectiveness of APR technology, but JML does not work while facing large projects like Defects4J. Xiong(2018) proposed PATCH-SIM and TEST-SIM. Their methods do not need oracle, classify the newly generated tests (test input) heuristically, and use the similarity of program execution paths to identify patches. It is too strict for the method using execution path similarity to identify overfitting patches. And it doe not further subdivide that overfitting patch. Inspired by the existing methods, this paper reinterprets PATCH-SIM and TEST-SIM by calculating the dynamic behavior expression of the program. It identifys patches heuristically, and further subdivides the overfitting patches.

3 PRELIMINARIES

To identify whether a patch is overfitting, PatchID relies on the following basic concepts.

3.1 Program State Abstraction

In order to describe the dynamic behavior of a program at runtime, this paper introduces the concept of program state abstraction (Chen et al, 2017, 2020). Program state abstraction is composed of program variables, basic expressions, extended expressions, and boolean expressions.

Variables. During the execution of each test, the values of these program variables are added into the set M_ℓ. These program variables are derived from two types of data.

- Exact values of numeric and boolean types
- Object identifier for a reference type expression

Expressions. In a program, reference types or numeric types can be monitored. These type of variables make up the basic expressions, which include (1) local variables (including parameters of M_{bug}) declared inside M_{bug}

(the method where the bug occurs), and these variables are visible at ℓ; (2) attributes of the class to which the M_{bug} belongs; (3)Any expression that can be evaluated at ℓ, but expressions with side effects such as increment, decrement, assignment, and creation cannot be monitored. In this paper, we use E_ℓ to denote the set of all monitorable basic expressions at ℓ (ℓ denotes the unique identifier of each statement). For each reference type r in the E_ℓ, the following two forms form the extended expression, including: (1) $r.f()$, $f()$ is a no-argument function and returns a type that can be monitored at ℓ. (2) When r is this, the attributes of r are monitorable at ℓ. In this paper, we use X_ℓ to denote the set of all extended expressions that can be monitored.

In order to explain the expression in detail, Figure 1 and Figure 2 show the buggy program and its corresponding patch. The buggy program functions as follows: The duplicate method receives two ArrayLists ($list1, list2$) and an int n. Its function is to copy n elements from $list2$ to $list1$.However, this program has an obvious error, which ignores the case that n is greater than the length of $list2$, When n is greater than the number of elements of $list2$, the program will throw Index Out Of Bounds exception. if statement is appended into the patch to avoid this error. *count* in the first line of the buggy program, $list1$, $list2$, and n in the arguments, are all basic expressions that can be monitored. At this point, the extended expressions for ℓ are $list1.size()$, $list2.size()$, and so on.

Boolean expressions. Boolean expressions are composed of relational expressions and logical expressions. In this paper, we denote by B_ℓ a boolean abstract set at ℓ. Boolean expressions can be expressed in the following four forms: (1)for each pair $k_1, k_2 \in E_\ell \cup X_\ell$ of expressions of the same type, B_ℓ includes $m_1 == m_2$ and $m_1 \neq m_2$; (2)for each pair $k1, k2 \in E_\ell \cup X_\ell$ of expressions of integer type, B_ℓ includes $k_1 \bowtie k_2$,for $\bowtie \in \{<, >, \leq, \geq\}$; (3)for each expression $b \in E_\ell \cup X_\ell$ of boolean type,B_ℓ includes b and $!b$; (4)for each pair $b_1, b_2 \in E_\ell \cup X_\ell$ of expressions of boolean type, B_ℓ includes $b_1 \vee b_2$ and $b_1 \wedge b_2$.

For example, in the first line of Figure 1, n and $list2.size()$ are both monitorable expressions, and both are of the same Integer type. Then the following six boolean expressions can be composed: $n > list2.size(), n >= list2.size(), n < list2.size(), n <= list2.size(), n == list2.size(), n! = list2.size()$.The boolean expression $n > list2.size()$ causes the program error. If the test satisfies this boolean expression, when the program enters the *while* loop, *count* will eventually equal $list2.size()$, which causes the program to throw an out-of-range error. So this results in test1 being a passing test and test2 being a failing test, because the size of $list2$ in test2 is less than 3.

3.2 PATCH-SIM and TEST-SIM

As mentioned earlier, a test suite is not like a formal specification in that its coding specifications are weak and incomplete. Low-quality test suites are a key reason why APR generates overfitting patches. To take full advantage of

```
1  public void duplicate(ArrayList<Integer> list1, ArrayList<Integer>
       list2, int n){
2    int count = 0;
3    while(count < n){
4      list1.add(list2.get(count));
5      count++;
6    }
7  }
```

Fig. 1 Buggy program

```
1  public void duplicate(ArrayList<Integer> list1, ArrayList<Integer>
       list2, int n){
2    int count = 0;
3    if(n > list2.size())return;
4    while(count < n){
5      list1.add(list2.get(count));
6      count++;
7    }
8  }
```

Fig. 2 Patch

```
public void test1(){
  ArrayList<Integer> list1 = new Arraylist();
  ArrayList<Integer> list2 = new Arraylist();
  list2.add(1);
  list2.add(2);
  duplicate(list1, list2, 1);
  assertEquals(list1.size(), 1);
}
```

Fig. 3 Passing test: test1

```
public void test2(){
  ArrayList<Integer> list1 = new Arraylist();
  ArrayList<Integer> list2 = new Arraylist();
  list2.add(1);
  list2.add(2);
  duplicate(list1, list2, 3);
  assertEquals(list1.size(), 0);
}
```

Fig. 4 Failing test: test2

existing and enhanced test suites for filtering overfitting patches, we need the following two definitions:

Definition 1 (TEST-SIM) When two tests have the same program behavior, then they will have the same test result, that is, the boolean expression b and the corresponding value are the same at the same statement, then both tests should be either a passing test or a failing test.

APR usually takes the patch that passes all tests as the correct patch, but the test suite cannot express a complete oracle. To enhance the test suite, we need to generate new tests, but PatchID's method of generating new tests does not and cannot know what the test output is. Only boolean expressions at run time are available. For example, there is a new test with $list1.size = 0$, $list2.size = 5$, and $n = 10$. Then the boolean expression of the test in the third line of the buggy program in Figure 1 is $n >= list2.size(), true$, which is the same as the boolean expression of test2 at this point, so this test is considered to be a failing test.

Definition 2 (PATCH-SIM.) With the correct patch, the passing test is the same as the previous boolean expression and its value, while the failing test should be different.

For example, in a buggy program, all passing tests result in a boolean expression b being false, and all failing tests result in b being $true$.Then determining whether the patch is overfitting is not just a single way to observe the output of the program, but by comparing the value of b in a statement before and after using the patch. The value of b should be consistent with the buggy program when the patch runs the passing test; it should be different from the buggy program when the patch runs the failing test. According to PATCH-SIM, we can see that in the previous example, patch is correct, because in the fourth line (corresponding to the third line of the buggy program), $n > list2.size()$ no longer exists, and all tests that make this boolean expression true will go to the *return* statement.

Through these two observations, we can dig out more information from a buggy program and a test suite, so that the identification of patches does not need oracle. New tests are heuristically classified by TEST-SIM, and patches are heuristically classified by PATCH-SIM.

4 APPROACH

This section mainly introduces the detailed process of PatchID, including its overview and three main modules, i.e. Snapshot finder, Test generation, and Identification.

4.1 Overview

Figure 5 below shows the overall flow of our approach.

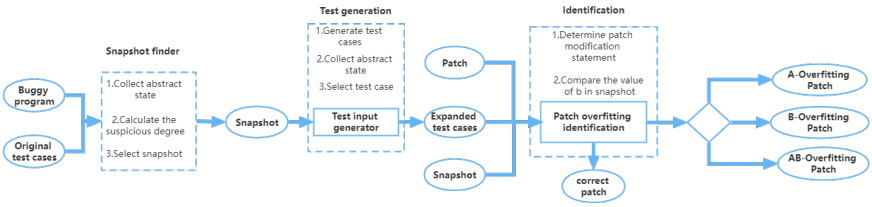

Fig. 5 Approach Overview

Take a buggy program, an original test set t_o, and the corresponding patch as input. This method firstly find the most suspicious snapshot s through Snapshot finder, and then generate new tests through s and Test input generator. Secondly, the new tests are added to the test suite to generate the extended test suite t_e. PatchID runs patches with t_e and saves the values of the

expressions in s. Finally, PatchID indentifies a overfitting patch by observing whether the expression value of each test case changes before and after using the patch. And the overfitting patches are subdivided further.

4.2 Snapshot Finder

Snapshot finder is designed to find the most suspicious snapshot s. This step has two functions. One is to provide a standard for generating new tests in the next step, so as to classify the new tests; the other is to compare the snapshot set obtained after running the test set for the patch in the third step in order to identify the type of the overfitting patch.

snapshot. snapshot s can be expressed as a 5-tuple $s = \langle \ell, b, ?, i, v_i \rangle$,where ℓ is the unique identifier of each statement, b is a boolean expression, ? is the value of b (true or false), i represents the unique serial number of each test in the test suite, and v_i represents the actual value of b of the test t_i during M_{bug} execution. Each snapshot has a corresponding degree of suspicion, which is determined by the following two factors: (1)a syntactic analysis of expression dependence ed_s; (2)a dynamic analysis dy_s. ed_s increases as the number of occurrences of b before and after ℓ; The more times ? corresponding b appears in failing tests and the less times ? corresponding b appears in passing tests, the greater the value of dy_s. The formula for calculating the degree of suspicion in this paper is defined as follow:

$$2/(ed_s^{-1} + dy_s^{-1}) \tag{1}$$

Because of the limitation of formula above for calculating the degree of suspicion, ? is not exactly the same as the result v of the actual execution of the test, so we must note that ? is used to classify new tests and v is used to classify patches. Snapshot is the core of this algorithm. This paper designs the following algorithm to construct Snapshot.

In the first step of this algorithm, PatchID creates a boolean expression set for each statement ℓ. Secondly PatchID uses t_o to run the buggy program. It first deletes the tests that do not cover M_{bug}, then saves the program abstraction state in the tests that cover M_{bug}, and calculates the value of each boolean expression in the set (real value v). A snapshot corresponding to each coverage test is generated. The third step is to calculate the suspicious degree of each snapshot. At this time, ? is obtained(PatchID will use this value to generate new tests). The fourth step is to select the most suspicious snapshot s. If there are multiple snapshots, select one of them at random. For the subsequent process, PatchID will save a snapshot set s_{bug}, whose ℓ,b,? as with s, i and v holds the real value for each test. In the previous example, $s = \langle 3, n >= list2.size(), true, i, v \rangle$ represents the boolean expression $n >= list2.size()$ in the third line of the buggy program, the error occurs because expression is evaluated to be true.

Algorithm 1 get the most suspicious snapshot

Require: t_o: original test suite; M_{bug}
Ensure: s_{bug}

```
1:  ex_b ← null
2:  function "GETSNAPSHOT"
3:      ex_b = createBooleanExpression(M_bug);
4:      for each t_i : t_o do
5:          if coverBugM(t_i) then
6:              compute(ex_b)
7:          else
8:              delete(t_i);
9:          end if
10:     end for
11:     s_bug = getSnapshot();
        return s_bug
12: end function
```

4.3 Test Generation

The goal of Test generation is to generate new tests using the TEST-SIM criterion. PatchID does not need to care about whether the output of these tests is correct, but it needs to know the value of the boolean expression b in the snapshot generated by these tests to be the same as ? of s. This means that PatchID requires some new failing tests that are used to enhance the test suite.

PatchID uses Evosuite(Fraser and Arcuri, 2011), an existing automated test generation tool, to generate a set of tests. Because Evosuite generates test cases for a class, the test cases should be selected. Similar to the steps in the Snapshot finder, select the test that covers the M_{bug} and save the program abstract state and snapshot. If the snapshot(ℓ, b, v) of the new test corresponds to the most suspicious $s(\ell, b, ?)$ respectively, then new failing tests will be added to the test suite t_e. PatchID does not select the new passing tests, because for the passing tests, M_{bug} not only outputs correctly, but also behaves correctly.

4.4 Identification

The purpose of this step is to determine if a patch is overfitting. It requires that snapshot set s_{bug}, extended test suite t_e, and the patch are provided previously. PatchID will run patch with the extended test suite, save the snapshot set s_{patch}, and then compare it with s_{bug} to determine the type of patch.

4.4.1 Select Statement

For buggy programs, patches typically include the following operations, that is, insert, delete, replace, and update. Then the ℓ in the snapshot of the buggy program cannot be directly monitored in the patch, because the position of

the statement has changed, and the statement ℓ_{patch} in the patch needs to be relocalized to monitor the same boolean expression b. We believe that no matter what kind of repair operation, the program can have correct program behavior only after the repair operation is completed, so the method in this paper selects the position of ℓ_{patch} in the next statement after the modification is completed. For some special cases, we also need to use other rules. We denote $start_s$ as the first statement that differs between the buggy program and the patch, and end_s as the last statement that differs. The rules for localizing $start_s$ and end_s are defined as follows:

- If $start_s$ is a block statement such as $for, while, if$, etc. and end_s is inside this block statement, then ℓ_{patch} is the next statement at the end of the block.
- If $start_s$ and end_s lie in sequential statements, ℓ_{patch} is the next statement in end_s.
- If end_s is the last statement of a program or a block of code, $\ell_{patch} = end_s$

4.4.2 Patch identification

After ℓ_{patch} is determined, PatchID will run patch using the test suite t_e to obtain the snapshot set $s_{patch} = \{\langle \ell_{patch}, b, ?, i, v_i \rangle \mid i = 0, 1, \ldots, n\}$. The steps for obtaining s_{patch} are similar to those in 4.2. The next step is to compare two snapshot sets s_{bug} and s_{patch} to determine whether the patch is overfitting. In order to identify overfitting patches, two variables N_f and N_p are needed to be calculated. For a failing test, N_f represents the number of same value of v between the two sets. For a passing test, N_p represents the number of different value of v between the two sets. Given the values of the variables N_f and N_p, the type of patch is defined as follows:

$$detection(p) = \begin{cases} A & N_f > 0 \bigwedge N_p = 0 \\ B & N_f = 0 \bigwedge N_p > 0 \\ AB & N_f > 0 \bigwedge N_p > 0 \\ correct & N_f = N_p = 0 \end{cases} \tag{2}$$

We map two snapshot sets one by one according to the unique identifier i of the test, and compare the v_i values of each test before and after using the patch. Depending on the type and source of tests, t_e is divided into three categories, that is passing, failing, and new. If $t_i \in t_{passing}$, N_p++ when v_i is different; if $t_i \in t_{failing} \cup t_{new}$, N_f++ when v_i is the same. After the two sets are compared, the values of N_f and N_p can be obtained. The type of the patch can be determined according to the above formula, N_f and N_p. For example, v corresponding to tese1 and test2 in the buggy program is false and true respectively. And both of their values in patch are false. Then $N_f = N_p = 0$, and patch is treated as correct.

5 EVALUATION

To experimentally evaluate the effectiveness of the PatchID, we propose the following six research questions:

- RQ 1: Can PatchID identify overfitting patches and classify correct patches generated by automated program repair? Does it have advantages over other methods?
- RQ2: How efficient is PatchID while identifing patches?
- RQ3: How reliable is PATCH-SIM based on boolean expressions?
- RQ4: What causes false positives and false negatives?
- RQ5: How useful is test generation for PatchID?
- RQ6: How reliable is PatchID for overfitting patch classification?

5.1 Dataset

In this paper, Patch is evaluated on two datasets. The one is the dataset collected in (Xiong et al, 2018), which is composed of the patches generated by six APRs on Defects4J. Another is Java+JML dataset created by Nilizadeh et al. **Defects4J.** At present, Defecets4j proposed by Just(2014) is the most widely used Java program dataset in the field of automatic program repair. Defects4J has 17 projects so far, which contain 835 defects. Each program defect in this dataset contains at least one test that can trigger it.This paper uses the six most frequently used projects in the dataset, namely, Chart, Time, Math, Lang, Closure and Mockito, where Chart is a project dedicated to displaying icons, Time is a project used for date and time processing, Math is a project for scientific computing, and Math is a project for displaying icons; Lang is a set of additional methods for manipulating JDK classes; Closure is an optimizing compiler for Javascript; Mockito is a mock framework for unit testing.The number of bugs contained in each project is shown in Table 1 below. In this paper, six existing repair tools are used to repair the Defects4J

Table 1 Defects4j Project evaluated

Project Name	Number of bugs
Chart	26
Time	26
Math	106
Lang	64
Closure	174
Mockito	38
Total	434

dataset, and candidate patches are obtained. The six APRs are jGenProg, Nopol 2015, Nopol 2017, ACS, HDRepair and jKali, respectively. jGenProg is the Java version of GenProg, which is a heuristic search repair tool based on genetic algorithm. Nopol is a technique for fixing conditional statement errors

in Java programs. It gives different repair strategies according to the type of error statement. If the location of the error code is a conditional statement, Nopol usually generates a repair patch to modify the original conditional statement; if the location of the error code is a non-conditional statement, it simply adds a new condition to skip the execution of the current statement. This paper includes two versions of Nopol 2015(Martinez et al, 2017) and Nopol 2017(Xuan et al, 2016). ACS is a conditional statement synthesis tool with high precisiton, which extracts patch templates for repair based on statistical analysis. HDRepair is also a repair tool based on statistical analysis. JKali is a re-implementation of Kali on Java, which is a repair tool to remove buggy statement only.

Java+JML dataset. This dataset proposed by Nilizadeh is the first proven publicly available dataset for Java programs. It consists of four parts, that is correct program, mutated wrong program, test suite, and APR-based patch. The program for this dataset has a JML specification for experimental evaluation. Java+JML dataset implements various classic algorithms and data structures, such as bubble sort, factorial, queue, and so on. They are both formally canonical small programs written in JML and, as such, can be thought of as programs with oracle. Test suites are created using an AFL-based fuzz tool, and are divided into Small and Medium based on the size of test suite. Error programs are created by injecting a single error into each Java program using PITest, a Java program mutation tool. PITest generates errors by changing the control condition, changing the assignment expression, removing the method call, and changing the return value. The APR-based repair patches are obtained using the following repair tools: ARJA-E, Cardumen, jGenProg, jKali, jMutRepair, Kali-a, and Nopol.

5.2 Experiment Setup

We implemented PatchID based on JAID. JAID is an automatic repair framework for generating patches for program bugs. It collects the program abstract state and uses boolean expressions as the criteria for verifying program bugs

RQ1. To evaluate the effectiveness of PatchID, we ran the collected patch set and saved the results of the patch identification and the values of the important variables in a file. These variables include the statement to be monitored, the expression and value of snapshot, passing test with different values, failing test with the same value, and new test.

RQ2. Record the running time of each patch. The time is calculated in whole minutes, and the extra seconds are rounded up and down.

RQ3. Manually analyze the values of N_f and N_p and the corresponding tests after each patch runs, and analyze the statistical information of each value.

RQ4. Manually analyze misclassified patches and analyze the reason why they failed.

RQ5. At this point, Evosuite usually finishes generating tests in about a minute and a half. Because Evosuite's strategy is to use fewer tests to cover as

many paths as possible, in practice, it can typically generate 20 $\sim$ 70 tests for a class. Through the first filter, there are usually only about ten tests covering M_{bug}. Due to the strict screening conditions, the number of tests that meet the requirements is usually only 0 $\sim$ 3 while further screening

RQ6. In order to verify the effectiveness of PatchID's classification method for overfitting patches, we manually verify the specific classification of overfitting patches. Because the Defects4J patch is too complex, this paper only analyzes it in the Java+JML dataset. The human identification is based on the different warnings generated from the warning file "esc.txt" corresponding to the buggy program and the warning file "ESC_Repaired.txt" of the patch in the dataset and the different codes between the two programs.

5.3 Experimental Result

5.3.1 Result of RQ1

Performance on Defects4J. A total of 220 patches are generated on the Defects4J dataset through APR tools. PatchID conducts experiments on these 220 patches to determine whether they are overfitting patches. 166 patches are run to determine whether they are overfitting patches, while the rest of the patches fail to give the final results because they exceed the set execution time limit. Except for 9 patches, 157 patches among the 166 patches, PatchID gives the results of whether they are overfitting patches. Results are shown in Table 2.

Table 2 The Results Identified by PatchID in Defects4j Dataset

Tools / Project	Chart	Closure	Lang	Math	Mockito	Time	Total
Nopol	12	39	10	26	1	8	96
jKai	5	0	0	9	0	1	15
jGenprog	6	0	0	13	0	1	20
HDRepair	0	0	2	6	0	1	9
ACS	1	0	4	11	0	1	17
Total	24	39	16	65	1	12	157

Table 3 and Table 4 show the running results of PatchID on related defect repair tools and different projects, respectively. As shown in the table, PatchID successfully filtered out 78 patches from 157 patches, including 63 overfitting patches and 15 correct patches. For 63 overfitting patches, PatchID successfully divided them into three categories, of which A-Overfitting Patches accounted for the most, reaching 50, followed by B-Overfitting Patches with 8. Number of AB-Overfitting Patches is 5.

Overfitting patch. From Table 4, we can find that PatchID works well on Nopol2015, Nopol2017, jKali and jGenprog (the worst accuracy is 50%), but it works poorly on ACS and HDRepair (the best accuracy is only 20%). We also found that among the overfitting patches generated by these six tools, the

Table 3 Performance On Different APRs

Tool	Correct	Overfitting	Correct identified	Overfitting identified	A	B	AB
Nopol2015	5	20	2(40%)	10(50%)	9	0	1
Nopol2017	3	68	2(66.66%)	36(52.94%)	25	8	3
HDRepair	4	5	3(75%)	1(20%)	1	0	0
ACS	11	6	7(63.63%)	1(16.66%)	1	0	0
jKali	1	14	0	8(57.14%)	8	0	0
jGenprog	6	14	1(16.67%)	7(50%)	6	0	1
Total	30	127	15(50%)	63(49.61%)	50	8	5

"Correct/overfitting identified" indicates the number of correct classifications from the "correct/overfitting" patch by PatchID.

A = A-Overfitting Patch,B = B-Overfitting Patch,AB = AB-Overfitting Patch

Table 4 Performance On Different Projects

Tool	Correct	Overfitting	Correct identified	Overfitting identified	A	B	AB
Lang	6	10	2(33.33%)	3(50%)	3	0	0
Math	16	49	8(50%)	22(44.90%)	20	1	1
Chart	3	21	(33.33%)	12(57.14%)	10	0	2
Time	2	10	2(100%)	6(60%)	5	1	0
Closure	2	37	1(50%)	20(54.05%)	12	6	2
Mockito	1	0	1(100%)	0	0	0	0
Total	30	127	15(50%)	63(49.61%)	50	8	5

patches that did not fix the original errors of the program were the most, and the patches that destroyed the original correct behavior of the program were relatively few. But Nopol2015 and Nopol2017 are tools that modify program conditional statements to fix bugs. A total of 12 patches destroyed the correct behavior of the program, while only jGenprog produced an AB-Overfitting Patch. We suspect that it is relatively easy to introduce new errors by modifying the conditional statements of the program. According to Project, the accuracy of overfitting patch identification is relatively stable, ranging from 43% to 60%. PatchID has the highest accuracy in Time Project, at 60%. The accuracy in Math is the lowest, only 44.90%. Here, the largest number of patches that destroy the original correct behavior of the program is Closure, with a total of 8.

Correct patch. Out of 157 patches, there are 30 correct patches, while PatchID can correctly classify 15 patches with an accuracy of 50%. This is exciting messages. As far as we know, there is no tool that has such a high accuracy accuracy. Among the patches generated by Nopol2017, HDRepair and ACS, the accuracy of PatchID is more than 60%, and the highest is 75%. From the perspective of Project, except Lang and Chart, the accuracy of other projects is not low. Of particular note is the 100% success rate on Mockito and Time projects.

Relative to Xiong's results on this dataset, we identified one more overfitting patch than Xiong's method, but Xiong's method did not identify any correct patch, while PatchID identified 15. For 220 patches, his method identified a total of 62 and PatchID identified 78 patches. However, Xiong improved the accuracy to 56.3% by pruning the average, and the PatchID is 49.7%. In addition, Xiong's method can only identify the patches of Chart, Lang, Math and Time, while PatchID involves the patches of six projects. PatchID is more extensive in its versatility.

Performance on Java+JML dataset. We selected 236 overfitting patches based on the Medium test suite and 336 overfitting patches based on the Small test suite from the Java+JML dataset. These overfitting patches are identified by the JML specification. In addition, there are 21 FalseNegatives patches (JML specification misidentify a correct repaired program as overfitted). The PatchID was run on a total of 593 patches, and the results of 380 patches were obtained, as shown in Table 5.

Table 5 Performance On Different Projects

PatchType	Collected	Validated
Medium	236	144
Small	336	221
FalseNegatives	21	15
Total	593	380

It can be seen from the data in Table 6 that PatchID can correctly identify 72 overfitting patches for Medium, with an accuracy of 50%; PatchID can identify 92 overfitting patches for Small, with an accuracy of 41.62%; In the patch of FalseNegatives, 5 correct patches were identified, and the accuracy is only 33.33%.

From the perspective of overfitting classification, PatchID does not identify any B-Overfitting patches on this dataset. Moreover, except for 4 AB-Overfitting patches, all other overfitting patches are of A-Overfitting type.

From the accuracy of Medium and Small, it is clear that as the number of tests in the test suite decreases, the success rate also decreases. This data shows that weak test suites can affect the accuracy of PatchID.

Table 6 Performance On Different PatchType

PatchType	Correct identified	Overfitting identified	A	B	AB
Medium	72(50%)	72(50%)	72	0	0
Small	129(58.37%)	92(41.62%)	88	0	4
FalseNegatives	5(33.33%)	10(66.67 %)	10	0	0
Total	206	174	170	0	4

5.3.2 Result of RQ2

Performance on Defects4J. We recorded the running time of 157 patches. As mentioned earlier, PatchID identifies a patch in three steps. And each step takes different amounts of time for different projects. As shown in Table 7, PatchID will get results in 5 minutes for most (57.32%) of the patches, in which the second step of Test Generation takes up most of the time. Patches that take more than five minutes to run take more time to get the abstract state of the program, because we have observed that Evosuite typically takes 1 to 2 minutes to generate new tests on Defects4J. In the experiment, there are three patches that take more than 60 minutes to run, one of which takes 123 minutes, and most of the time is spent running tests. The average time is 11.6 minutes to identify whether a patch is overfitting, and 76.42% of the patches are within the acceptable range of t < 10 min, which is satisfactory.

Table 7 Running time distribution of different projects

Project	t <= 5	t <= 10	t <= 30	t > 30
Chart	22	2	0	0
Lang	12	2	1	1
Math	37	12	6	10
Time	8	0	3	1
Mockito	1	0	0	0
Closure	10	14	10	5
Total	90(57.32%)	30(19.10%)	20(12.73%)	17(10.83%)

The experimental results of the memory occupied by the algorithm are shown in Figure 6. Among the 157 patches, 145 patches use less than 512MB of memory to obtain the results, another 10 patches use less than 1024MB of memory, and only 2 patches use more than 1024MB of memory. The patch that uses the most memory is still the one in Closure project, which uses 1405 MB of memory, and the other in Math project, which uses 1044 MB of memory.

Figure 7 shows the average amount of memory used by each project. The average memory used for Mockito is the least, with only 50 MB used. Others are Time, Lang, Math, Chart, Closure in descending order. The average memory used for Closure is 349MB. From our experimental data, Closure is the most time-consuming and memory-consuming, which means that our method consumes more resources in terms of Javascript optimizing compiler.

Performance on Java+JML dataset. We recorded the distribution of the time spent of the 380 patches and the average amount of memory they used. From Table 8, we can see that 378 patches can produce results within 10 minutes, and only 2 patches exceed 10 minutes. From Figure 8, Medium patch uses the most memory, at 99 MB. The smallest is FalseNegatives, which is only 62 MB. However, none of these three types of patches exceed 100MB.

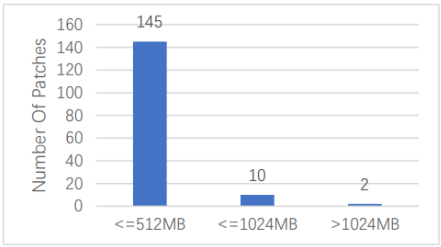

Fig. 6 Distribution of patches in memory used

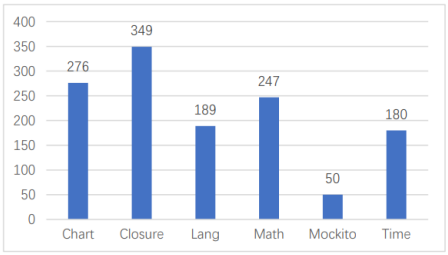

Fig. 7 Average memory used by the project

Table 8 Number of patches at running time interval for different PatchType

PatchType	t <= 10	t > 10
Medium	143	1
Small	200	1
FalseNegatives	15	0
Total	378	2

5.3.3 Result of RQ3

Performance in Defects4J. We observed the values of N_f and N_p from 63 overfitting patches, and we obtained the following two tables. It can be seen from Figure 7 that the number of patches with $N_f = 1$ is the largest, 47. Most of the patches have N_f values between 0 and 2. This shows that it is very difficult to identify a patch simply by the test output, because when the patch has only one program behavior error, its error is not reflected in the test output. PatchID, on the other hand, focuses on the behavior of the program, and as long as the wrong behavior of the program is exposed by a failed test, then the patch can be identified as overfitting.

Table 9 shows N_p values for each patch with a regression error. We find that more than half of the patches have a large N_p, with the maximum reaching 764. The patch note for this maximum breaks a very important correct behavior in the original program, which causes such a large number of passing tests

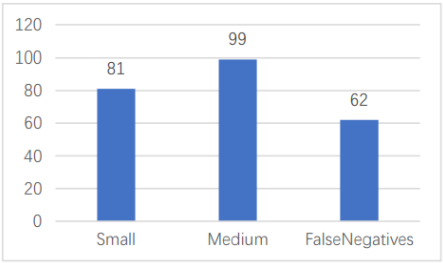

Fig. 8 Average memory used by the PatchType

to behave incorrectly. It is worth mentioning that from the distribution of Project, Closure still accounts for the majority of the number.

Generally speaking, PATCH-SIM works better, and it can effectively distinguish the three types of overfitting patches. And we got an interesting conclusion: on the one hand, the overfitting patch can fix the wrong behavior of the original program to some extent, but it still can not completely fix its errors; on the other hand, once the overfitting introduces the regression error, the regression error will have a greater impact on the original correct behavior of the program.

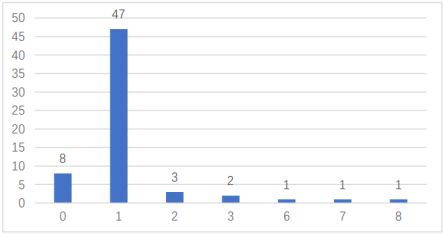

Fig. 9 Number of patches per N_f value

Performance in Java+JML dataset. Table 10 shows N_f and N_p results on the Small and Medium datasets of Java+JML dataset. It can be seen from the table that most of the patches have $N_f <= 10$, accounting for 90%, because most of the patches in the above dataset are overfitting. N_f and N_p results of four AB-Overfitting patches is listed in Table 11, all of which are from the small type. From the table, we can see that their N_p is not more than 2.

5.3.4 Result of RQ4

We manually analyzed all the patches that were incorrectly identified by PatchID, and we believe that the following two points are the reasons for the wrong results.

- Weak test suite
- Unsatisfying snapshot find strategy

Table 9 Patches with regression fault

Patch	Project	N_P
Patch105	Closure	764
Patch112	Closure	14
Patch119	Closure	19
Patch137	Closure	41
Patch138	Closure	53
Patch141	Closure	9
Patch147	Closure	1
Patch177	Math	1
Patch180	Time	14
Patch32	Math	5
Patch4	Chart	1
Patch88	Chart	1
Patch94	Closure	2

Table 10 Distribution of N_f

PatchType	$N_f <= 10$	$N_f > 10$
Small	89	3
Medium	59	13

Table 11 AB-Overfitting Patch

Patch	N_f	N_p
example240	1	2
example241	1	2
example308	10	1
example50	1	2

These two reasons together lead to the wrong identification of PatchID. On the one hand, most of the test suites have only $1 \sim 2$ failing tests, and there are too few tests covering M_{bug}, which makes it less likely that PatchID can actually collect the correct boolean expression. On the other hand, although PatchID can generate a large number of snapshots, it can cause multiple snapshots to be equally suspicious for the first reason. The reason for the same suspicious degree is the defect of the suspicious degree formula. In addition, since only one snapshot can be selected as the criterion for the patch identification, PatchID only randomly selects one of them, and there is a considerable probability with an unrelated snapshot selected, which will also lead to the failure of the identification.

In addition, because we only selected new failing tests in the Test generation, our experiment paid more attention to whether the patch fixed the original error, but ignored whether the patch introduced new errors, although the experimental results showed that only 13 patches introduced new errors.

5.3.5 Result of RQ5

In terms of generating tests, we use the existing Evosuite tool to generate tests. For each patch, Evosuite takes approximately 1∼2 minutes to generate tests. Because Evosuite is to cover as many paths as possible with as few tests as possible, which results in fewer new tests originally covering M_{bug} that satisfy the boolean expression in the snapshot. But during the experiment, the new test has no effect on N_f. We looked at the program, and of course it's possible that Evosuite didn't generate a test that satisfied the snapshot. As mentioned in RQ3, we should add the passing tests in the generated tests to the test suite, which should theoretically enhance the role of Test generation.

5.3.6 Result of RQ6

Table 12 shows the effect of PatchID on Java+JML dataset for overfitting patch classification. We find that the number of A-Overfitting patches and AB-Overfitting patches are similar, which shows that APR patches can actually introduce a large number of regression errors. In addition, the number of AB-D is much larger than that of AB-S, which also shows that regression errors generally occur in methods other than M_{bug}. According to the classification results of PatchID for overfitting patches, the number of correctly classified patches on Small and Medium datasets is 40 and 25 respectively, and All these patches are A-Overfitting patches.

It can be seen from Table 12 that for most AB-Overfitting patches, PatchID will identify them as A-Overfitting patches, which is due to the limitations of PatchID. Because PatchID only verify the program abstract state of a single method M_{bug}, Most of the regression errors occur in methods other than M_{bug}, which is why there are so many AB-D patches. However, according to Table 13, PatchID is quite accurate in identifying the patches that do not fix the original errors. There are 163 patches (including A, AB-D, and AB-S) that do not fix the original errors, including 91 for Small and 72 for Medium. PatchID can identify all these patches that do not fix the original errors, with a success rate of 100%.

Table 12 Classification Of Overfitting Patches

PatchType \ Overfitting	A	B	AB-D	AB-S	Total
Small	44(40)	1	41	6	92(40)
Medium	25(25)	0	43	4	72(25)

The number of manually identified patches of each category and the number of patches of each category correctly classified by PatchID are given in Table 12. The number in parentheses represents the number of patches correctly identified by PatchID. AB-D indicates the number of patches for methods where regression errors occur outside of the M_{bug}, and AB-S indicates the

number of patches where regression errors still occur in the M_{bug}. For example, if the method X calls the M_{bug}, and the patch modifies the M_{bug} to cause an error in the program behavior of the method X, then we consider this patch an AB-D Overfitting Patch.

Table 13 Classification Of A Patches

PatchType \ Overfitting	A-Patch
Small	91(91)
Medium	72(72)

A-Patch includes A-Overfitting patch and AB-Overfitting patch.

6 CONCLUSION

In this paper, an overfitting identification method is proposed, which combines program expression and program similar behavior, and overfitting patches are further divided into three categories. We implement the method PatchID proposed in this paper on the JAID framework. As we have seen in the experiment, the PatchID method is able to effectively filter out 78 patches from 157 patches for Defects4J dataset and 169 patches from 380 patches for Java+JML dataset. The experimental results show that we should dig deeper into the information given by the test suite, not just limited to test input and test output. Future work will focus on: (1) strengthening the scope of program monitoring to further improve the success rate of overfitting patch classification, (2) providing an effective repair scheme for the identified overfitting patches to make them closer to the correct patches.

References

Afzal A, Motwani M, Stolee KT, et al (2019) Sosrepair: Expressive semantic search for real-world program repair. IEEE Transactions on Software Engineering 47(10):2162–2181

Chen L, Pei Y, Furia CA (2017) Contract-based program repair without the contracts. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 637–647

Chen L, Pei Y, Furia CA (2020) Contract-based program repair without the contracts: An extended study. IEEE Transactions on Software Engineering 47(12):2841–2857

Durieux T, Cornu B, Seinturier L, et al (2017) Dynamic patch generation for null pointer exceptions using metaprogramming. In: 2017 IEEE 24th

International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 349–358

Falleri JR, Morandat F, Blanc X, et al (2014) Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pp 313–324

Forrest S, Nguyen T, Weimer W, et al (2009) A genetic programming approach to automated software repair. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, pp 947–954

Fraser G, Arcuri A (2011) Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp 416–419

Gao F, Wang L, Li X (2016) Bovinspector: automatic inspection and repair of buffer overflow vulnerabilities. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp 786–791

Gao Q, Xiong Y, Mi Y, et al (2015) Safe memory-leak fixing for c programs. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, IEEE, pp 459–470

Ghanbari A, Benton S, Zhang L (2019) Practical program repair via bytecode mutation. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp 19–30

Hu Y, Ahmed UZ, Mechtaev S, et al (2019) Re-factoring based program repair applied to programming assignments. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 388–398

Hua J, Zhang M, Wang K, et al (2018) Towards practical program repair with on-demand candidate generation. In: Proceedings of the 40th international conference on software engineering, pp 12–23

Ji T, Chen L, Mao X, et al (2016) Automated program repair by using similar code containing fix ingredients. In: 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), IEEE, pp 197–202

Jiang J, Xiong Y, Zhang H, et al (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, pp 298–309

Just R, Jalali D, Ernst MD (2014) Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 international symposium on software testing and analysis, pp 437–440

Ke Y, Stolee KT, Le Goues C, et al (2015) Repairing programs with semantic code search (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 295–306

Kim D, Nam J, Song J, et al (2013) Automatic patch generation learned from human-written patches. In: 2013 35th International Conference on Software Engineering (ICSE), IEEE, pp 802–811

Kim J, Kim S (2019) Automatic patch generation with context-based change application. Empirical Software Engineering 24(6):4071–4106

Koyuncu A, Liu K, Bissyandé TF, et al (2019) ifixr: Bug report driven program repair. In: Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 314–325

Le XBD, Lo D, Le Goues C (2016) History driven program repair. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), IEEE, pp 213–224

Le XBD, Chu DH, Lo D, et al (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, pp 593–604

Lee J, Hong S, Oh H (2018) Memfix: static analysis-based repair of memory deallocation errors for c. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 95–106

Marginean A, Bader J, Chandra S, et al (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, pp 269–278

Martinez M, Monperrus M (2016) Astor: A program repair library for java. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp 441–444

Martinez M, Durieux T, Sommerard R, et al (2017) Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. Empirical Software Engineering 22:1936–1964

Mechtaev S, Yi J, Roychoudhury A (2015) Directfix: Looking for simple program repairs. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, IEEE, pp 448–458

Mechtaev S, Yi J, Roychoudhury A (2016) Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th international conference on software engineering, pp 691–701

Mechtaev S, Gao X, Tan SH, et al (2018a) Test-equivalence analysis for automatic patch generation. ACM Transactions on Software Engineering and Methodology (TOSEM) 27(4):1–37

Mechtaev S, Nguyen MD, Noller Y, et al (2018b) Semantic program repair using a reference implementation. In: Proceedings of the 40th International Conference on Software Engineering, pp 129–139

Nguyen HDT, Qi D, Roychoudhury A, et al (2013) Semfix: Program repair via semantic analysis. In: 2013 35th International Conference on Software Engineering (ICSE), IEEE, pp 772–781

Nilizadeh A, Leavens GT, Le XBD, et al (2021) Exploring true test overfitting in dynamic automated program repair using formal methods. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE, pp 229–240

Qi Z, Long F, Achour S, et al (2015) An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp 24–36

Rothenberg BC, Grumberg O (2016) Sound and complete mutation-based program repair. In: FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings 21, Springer, pp 593–611

Saha S, et al (2019) Harnessing evolution for multi-hunk program repair. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp 13–24

Tan SH, Yoshida H, Prasad MR, et al (2016) Anti-patterns in search-based program repair. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp 727–738

Tian Y, Ray B (2017) Automatically diagnosing and repairing error handling bugs in c. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp 752–762

van Tonder R, Goues CL (2018) Static automated program repair for heap properties. In: Proceedings of the 40th International Conference on Software Engineering, pp 151–162

Wang Y, Chen Y, Shen B, et al (2017) Crsearcher: Searching code database for repairing bugs. In: Proceedings of the 9th Asia-Pacific Symposium on Internetware, pp 1–6

Wen M, Chen J, Wu R, et al (2018) Context-aware patch generation for better automated program repair. In: Proceedings of the 40th international conference on software engineering, pp 1–11

Xin Q, Reiss SP (2017a) Identifying test-suite-overfitted patches through test case generation. In: Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis, pp 226–236

Xin Q, Reiss SP (2017b) Leveraging syntax-related code for automated program repair. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 660–670

Xiong Y, Wang J, Yan R, et al (2017) Precise condition synthesis for program repair. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, pp 416–426

Xiong Y, Liu X, Zeng M, et al (2018) Identifying patch correctness in test-based program repair. In: Proceedings of the 40th international conference on software engineering, pp 789–799

Xuan J, Martinez M, Demarco F, et al (2016) Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Transactions on Software Engineering 43(1):34–55

Yan H, Sui Y, Chen S, et al (2016) Automated memory leak fixing on value-flow slices for c programs. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, pp 1386–1393

Yang B, Yang J (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, pp 1–8

Yang J, Zhikhartsev A, Liu Y, et al (2017) Better test cases for better automated program repair. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp 831–841

Yu Z, Martinez M, Danglot B, et al (2019) Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. Empirical Software Engineering 24:33–67

Yuan Y, Banzhaf W (2018) Arja: Automated repair of java programs via multi-objective genetic programming. IEEE Transactions on software engineering 46(10):1040–1067

Yuan Y, Banzhaf W (2020) Toward better evolutionary program repair: An integrated approach. ACM Transactions on Software Engineering and Methodology (TOSEM) 29(1):1–53