Master's Thesis

Filtering Out Incorrect Patches with Generalized Tests

Md Mazba Ur Rahman

Department of Computer Science and Engineering

Ulsan National Institute of Science and Technology

2022

Filtering Out Incorrect Patches with Generalized Tests

Md Mazba Ur Rahman

Department of Computer Science and Engineering

Ulsan National Institute of Science and Technology

Filtering Out Incorrect Patches with Generalized Tests

A thesis/dissertation submitted to
Ulsan National Institute of Science and Technology
in partial fulfillment of the
requirements for the degree of
Master of Science

Md Mazba Ur Rahman

06/16/2022

Approved by

Advisor

Jooyong Yi

Filtering Out Incorrect Patches with Generalized Tests

Md Mazba Ur Rahman

This certifies that the thesis of Md Mazba Ur Rahman is approved.

06.16.2022

Signature

Advisor: Jooyong Yi

Signature

Mi-jung Kim

Signature

Yuseok Jeon

Signature

Abstract

Automated program repairing has been growing rapidly in the last years. There are many approaches for using these techniques and one of them is generate-and-validate. Generate-and-validate or, G&V has become very popular. When the APR tool generates patches, overfitting patches are also included there. That means, those patches are actually not correct patches but they pass all the tests available in the test suite. This overfitting problem is a big challenge in APR still now. So the goal of the G&V approach is eliminating the overfitting patches after patches are generated by the APR tool. This work introduces a novel lightweight specification methodology that uses an existing failing test. So the idea is that if developers generalize the existing failing tests by expressing their insight about the bug, then many incorrect patches (generated by the APR tools) in the list can be filtered out and the user has to review only the remaining patches. Our approach can be particularly useful when an APR tool returns a list of plausible patches to the user, as done in multiple recent APR tools. Many incorrect patches in the list can be filtered out using a single generalized test, and the user only needs to review the remaining patches. Our additional experiment results on a state-of-the-art APR tool, JAID, show that this usage scenario is promising, filtering out hundreds of incorrect patches while keeping all correct patches.

Contents

List of figures

List of Tables

CHAPTER 1. INTRODUCTION

In the generate-and-validate approach at first a patch candidate Pc is generated by the APR tool and then it is validated. In the validation step a user-given test-suite is used. As we know, simply passing all the tests does not mean that the patch is a correct patch. A patch can pass all the tests without being correct. If Pc passes all tests in the test suite then it is called a plausible patch. Previously the APR tools used to generate only one patch , which is the first patch that passes all the tests available. But if we simply take the first patch then there is a high chance that the patch is actually an overfitting patch. Considering this fact we have to actually generate a list of plausible patches so that we do not miss a correct patch that exists in the patch space. This is done in several APR systems lately [4, 10, 11, 37]. After getting the list of plausible patches, the user has to inspect them one by one to find out the correct patch. This manual inspection part has not been analyzed much.

This work proposes a new lightweight mechanism, *PORACLE*, that allows developers to generalize the failing tests using "preservation condition". From a list of plausible patches it has the ability to filter out the incorrect ones, leveraging some input from the developer.

The following conditions must be met for this strategy to be successful. Firstly, as many incorrect patches as possible should be discarded by the tool, which means we require a high level of recall. Second, and more crucially, right patches should not be dismissed because right or correct patches are rare [21] and if one correct patch is discarded then it will be very hard to find another one in the list of plausible patches, which necessitates almost perfect precision. Finally, the proposed specification approach must be simple to implement.

Several patch categorization (PC) algorithms [10, 33, 39, 42, 43] have recently been proposed for determining patch validity. Score-based approaches [33, 39, 43] and evidence-based approaches can be used to categorize these PC strategies. The previous techniques use information like distributed representations of patches and path spectra of patched and pre-patched (i.e., defective) versions to derive a patch score. To make a classification judgment, the computed score of a patch is compared to a predefined threshold. Deciding a threshold that allows for high recall while maintaining precision near 100 percent is usually difficult.

Similarly, differential testing [24, 29, 30] is applied in evidence-based techniques [10, 41] to find

behavioral differences across buggy/original and patched versions. The detected behavioral difference is strong proof that the patch is wrong, when the output of the buggy version is valid. For a good recall rate, we should know about the correctness of the output from the original version. Here, perfect precision can be obtained because an invalid patch can only be discarded if there is clear evidence of error. These methods, however, have a low recall rate at the moment. The reason for this problem is that it's tough to tell whether the output seen in the original version is correct without input from the user, and current models only filter out those fixes which cause crashes.

CHAPTER 2. Background and related work:

2.1 Automated program repair: Automated program repair tool is a tool that can automatically generate patches. Facebook is already starting to use this kind of tool internally. The input to the APR tool is a buggy program and a test suite or a set of test cases. Among those tests, one will be a failing test and that is how we know that the program is buggy. So, when we pass a buggy program and a set of tests then the tool will generate a patched program that will pass all the tests we have.

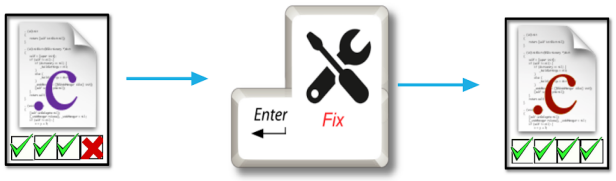


Figure 1: Test driven automated program repair.

2.2 Overfitting Problem: The overfitting problem [31], which happens because numerous incorrect patches exist in the patch space and are often not removed by the particular test suite [21, 31], is one of the main difficulties in APR. Many APR systems include either patch ranking algorithms, or score-based patch classification, or evidence-based patch classification to solve this problem.

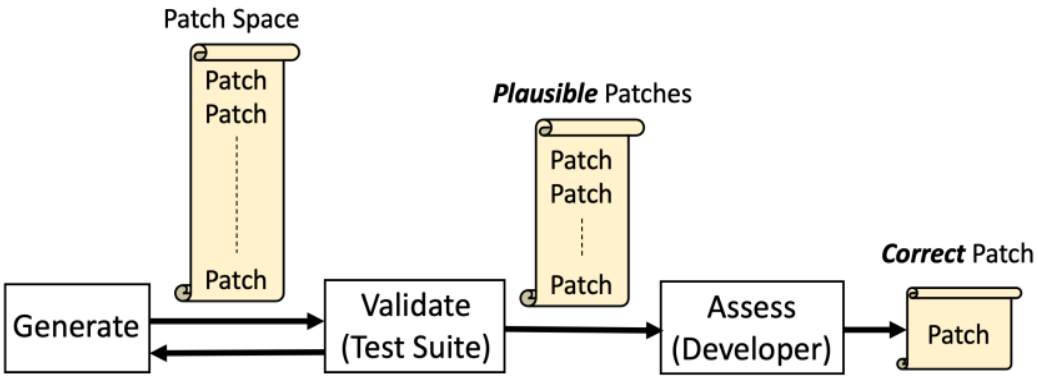


Figure 2: Three steps used in recent generate-and-validate (G&V) APR tools generating a list of plausible patches, among which the developer should manually look for a correct patch.

2.3 Ranking algorithm and Score-based Patch Classification approach: To combat against the overfitting problem, APR techniques frequently incorporate patch ranking algorithms. Patch ranking is based on identifying patches that are more likely to be correct and ranking them higher than patches that are more likely to be incorrect. Prophet [20], for example, uses a probabilistic model learnt from existing patches to rank patch candidates. Patch ranking is also performed by other APR tools such as CapGen [36], ACS [40], SimFix [15], and JAID [4] in the same way. The Patch Classification technique's aim is to preserve correct patches while filtering out incorrect ones. Score-based PC approaches calculate and utilize the scores of the patches to classify them. A simple binary scoring technique is applied in anti-patterns (common patterns of incorrect patches) [32]; patches adhering to anti-patterns receive a low score and are rejected. Patches are accepted in all other cases.

2.4 Patch evaluation. Patch classification (PC) techniques (including Poracle) are distinguished from patch evaluation (PE) tech- niques [38, 41, 44, 47] where patch correctness is evaluated based on correct versions in the benchmark. PC techniques do not assume the existence of correct versions.

CHAPTER 3. Our approach:

We introduce a semi-automatic patch-classification method that consists of two parts: generalizing an existing failing test with preservation conditions and differential fuzzing.

3.1 Preservation Invariants: To filter out incorrect patches, we don't think it's necessary to generalize an assertion. We propose instead that we apply preservation invariants (i.e., Formula (2)). We describe four possible preservation invariant patterns in this part, in order of coverage, that cover all 77 problematic versions in our dataset (see Section 5.1.1). Table 1 shows the distribution of the four patterns. We generalize a failing test from the Defects4J benchmark [17] for each version. The concept of our preservation invariant is motivated by software change contracts [45] and mutual summaries [12] where the user's change intention is expressed as a program contract. Whereas these techniques require the users to write a separate contract, our approach enables the users to directly and slightly edit an existing failing test familiar to them. More recently, program contracts have also been used to verify the correctness of APR-generated patches using formal verification techniques [23]. To perform verification, full specifications for all methods in the program should be written, and to mitigate this difficulty, [23] is applied to only small programs (e.g., IntroClass- Java [8]). Compared with these approaches, our approach is more lightweight.

When we have a patch and we have the original or buggy version of the program then we can apply the patch and get the patched version. Then we can run both the buggy version and the patched version and check if we can find any output difference between the patched version and the buggy version, for a specific input. For finding that input we have to use differential fuzzing.

The overfitting problem of APR occurs due to the fact that a failing test used by an APR system expresses only very limited information about change intention. While passing tests can reveal additional user intention, those passing tests are prepared without considering a bug at hand. This results in a situation where a user of an APR system should rely on luck for rejecting incorrect patches. There is no wonder why APR systems suffer from the overfitting problem! Instead of passively using tests prepared without considering them to be used for APR, what if a developer more actively expresses what developers want from an APR system that finds out how user's change intention can be achieved?

So, when we detect an output difference between the two versions then we should know if the output from the buggy version is the correct output or not. To determine this correctness we use preservation conditions. Preservation condition is the condition under which the buggy version will return the same output as the fixed version. The condition will be written by the developer and here he/she will express his/her intention. After using the preservation condition, if we see a difference in the outputs for a specific input then we can definitely reject the patch (because after using the preservation condition we do not expect any difference between the buggy version and the patched versions outputs if the patch is a correct patch). This difference is the evidence of the incorrectness of the patch.

In this approach we utilize an existing failing test. We extend the failing test and equip it with a preservation condition. For writing preservation conditions we have found four patterns which we will describe in the next section. Using those four patterns we can write preservation conditions for more bug versions. We have run experiments on 79 bug versions which include more than 300 patches. We create a differential fuzzer that accepts Java programs natively and controls differential fuzzing with a state-propagation metric.

We propose a novel specification methodology consisting of the following three phases.

Phase 1: Identifying a failing test. Instead of using a white-box approach, we advocate using a black-box approach for easy usability. Specifically, we suggest generalizing a failing test, a mandatory input to a test-driven APR system.

Phase 2: Parameterizing a failing test. To generalize the input of the existing test, we parameterize the original test (constant values appearing in the test code are parameterized) and mutate parameters using a fuzzer. An obtained parameterized test can be perceived as a parameterized unit test (PUT) [34] or a property- based test (PBT) [6, 13].

Phase 3: Generalizingan oracle into a preservation invariant. Given a parameterized test T($\vec{x}$) where $\vec{x}$ represents parameters, specifying an oracle function ψ ($\vec{x}$) returning expected output for $\vec{x}$ is not trivial. Even if a developer has a perfect understanding of the program under testing, it is often difficult to express her knowledge as a function ψ that satisfies the following:

$$\forall \vec{v} : T(\vec{v}) = \psi(\vec{v}) \qquad (1)$$

where T ($\vec{v}$) represents the output of test T when inputs $\vec{v}$ is assigned to parameters $\vec{x}$. Although writing an oracle function can be avoided by specifying a property that should always hold (such as decrypt (encrypt (x)) = x), it is usually not easy to find such a metamorphic relation that is effective at bug finding [5, 23]. Instead of specifying ψ, we suggest an alternative specification construct φ we call a preservation condition. Instead of specifying ψ, we suggest an alternative specification construct φ we call a preservation condition. Given a parameterized test T($\vec{x}$) applied to a buggy program P and a correctly patched program P' , preservation condition φ ($\vec{x}$) should satisfy the following:

$$\forall \vec{v} : \varphi(\vec{v}) \Rightarrow T_P(\vec{v}) = T_{P'}(\vec{v}) \qquad (2)$$

where $T_P(\vec{v})$ and $T_{P'}(\vec{v})$ represent the output of P and P' , respectively, when inputs $\vec{v}$ is assigned to parameters $\vec{x}$. Here the Intention is that for input *I* that satisfies φ, the output should be preserved between pre-patched and patched versions. Conversely, if output difference is observed for *I*, that evidences the incorrectness of P'. We call formula (2) a preservation invariant.

Overall, we make the following contributions in this work:

• A specification methodology for patch validation: We show for the first time how an existing test can be generalized to filtering out incorrect patches. Our lightweight specification method allows developers to keep using their familiar interface for specification– i.e., test code. We apply a proposed specification methodology to the failing tests of 77 buggy versions in our dataset. All 77 generalized tests are available in our replication package.

• Empirical findings: One concern of using our black-box spec- ification is that, while performing differential fuzzing, it can be difficult to observe a state difference between the versions since internal state differences occurring in the production code are not necessarily propagated to the test code. To assess this concern, we run a differential fuzzer over the 458 patches in our dataset. We found that in fact, many incorrect patches are in the vicinity of the original failing test; many incorrect patches are detected quickly (in about 30s). However, when preservation invariants are removed, many correct patches are wrongly filtered out, suggesting the im- portance of specification. Compared with four state-of-the-art PC techniques [33, 39, 42, 43], our approach shows better or comparable performance, showing high recall and 99% precision.

• Reduction in manual patch-review effort: Our work is motivated by the high cost of assessing a large number of plausible patches. By filtering out incorrect patches, the user only needs to review the remaining ones. To evaluate this usage scenario, we apply our approach to ranked lists of plausible patches generated from a state-of-the-art APR tool, JAID [4]. Our experimental results show that the number of patches to review significantly decreases after using our approach (on average, 108

patches/version), while all correct patches are retained. Note that a single generalized test is used to validate all plausible patches in a ranked list.

• User study: We use a preservation condition in a test based on the intuition that it would be easy to write a preservation condition. To assess our intuition, we conduct an end-to-end user study with 66 participants. Our participants tended to find out the correct patch more easily using our approach than with the manual one.

• Replication package: We provide a replication package through github. Our package contains 77 generalized tests and our custom fuzzer that supports our specification APIs.

3.2 Generalized tests

The generalized tests we used in our experiments are available in the deltas directory. In each sub-directory, we use the following name convention "[Project]_bug[bug_id]" such as "Math/Math_bug2". Using a QuickCheck framework [6] (junit-quickcheck [13]) we obtain various random values for the parameters to perform differential fuzzing. By preservation condition the users can express their intention about which behavior should be preserved using the following two APIs we provide: (1) logOutIf and (2) ignoreOutOfOrg. We have four different patterns of preservation invariants. Such as;

```
1  public void testSmallDegreesOfFreedom(double d1,
2                  double d2, double d3)
3    try {
4      FDistributionImpl fd = new FDistributionImpl(d1, d2);
5      double p = fd.cumulativeProbability(d3);
6      double x = fd.inverseCumulativeProbability(p);
7      Log.logOutIf(/* preservation condition */true,
8          /* outputs to compare */ () -> new Double[] {x});
9    } catch (Exception e) {
10     // original (pre-patched) version: ignore
11     // patched version: log a predefined message
12     Log.ignoreOutOfOrg();
13   }
14 }
```

Figure 3: A generalized test for Math95 (UE)

3.2.1 UE (Unexpected Exception): We enclose the original test code inside a try-catch block to ignore abnormal termination of the buggy version. In this case we use "true" as the preservation condition.

```
1  public void testGcd(int i, int j) {
2    /* Original body:
3    try {
4      MathUtils.gcd(Integer.MIN_VALUE, 0);
5      fail("expecting ArithmeticException");
6    } catch (ArithmeticException expected) { // expected } */
7    try {
8      final long actual = MathUtils.gcd(i, j);
9      boolean complement = !( (i==Integer.MIN_VALUE && j==0)
10         || (i==0 && j==Integer.MIN_VALUE) );
11     Log.logOutIf(complement, () -> new Long[] { actual });
12   } catch (ArithmeticException e) {
13     Log.logOutIf(!complement, () -> new String[] { e.toString() });
14   } catch (Exception e) { Log.ignoreOutOfOrg(); }
15 }
```

Figure 4: A generalized test for Math99 (CC)

3.2.2 CC (Complementary Cases): Behavioral preservation is enforced for the inputs complementary to the original test input.

A software fault often occurs when corner-case behavior is not yet implemented. Consider Figure 4 where the original test body is shown in lines 3–6. The bug occurs because method gcd fails to handle a corner case input cor- rectly; when gcd takes as input a pair of Integer.MIN_VALUE and 0, an ArithmeticException is expected to be thrown, but the buggy version fails to do so. In this case, developers would want to test whether program behavior is preserved in the complementary cases where input is not a pair of Integer. MIN_VALUE and 0. The generalized test shown in Figure 4 can detect an incorrect patch that returns an incorrect output (line 11). It can also detect patches that fail to throw an expected ArithmeticException (line 13) or throw an unexpected exception (line 14).

```
1  public void testSSENonNegative(double d1, double d2
2    double d3, double d4, double d5, double d6) {
3    try {
4      double[] y = { d1, d2, d3 }; double[] x = { d4, d5, d6 };
5      SimpleRegression reg = new SimpleRegression();
6      for (int i = 0; i < x.length; i++) { reg.addData(x[i], y[i]); }
7      double ret = reg.getSumSquaredErrors();
8      // Original: assertTrue(ret >= 0.0);
9      Log.logOutIf(ret >= 0.0, () -> new Double[] { ret });
10   } catch (Exception e) { Log.ignoreOutOfOrg(); }
11 }
```

Figure 5: A generalized test for Math105 (EA)

3.2.3 EA (Existing Assertion): Often, existing tests already contain general assertions. We use an existing assertion condition as a preservation condition. Consider Figure 5 where the original test uses an assertion condition, ret >= 0.0 (line 8). Since it is clear that target method getSumSquaredErrors should return a non-negative value (i.e., ret >= 0.0), we reuse the existing oracle condition as a preservation condition (line 9). Note that reusing the existing condition as a preservation condition does not mean that oracle power stays the same. An incorrect patch may satisfy ret >= 0.0, even when it changes the correct output of the original version. Such incorrect patches can be detected with our test (since outputs will differ between the two versions), whereas the original oracle using the same condition fails to detect the same.

```
1  public void testLang300(int n, int m) {
2     // NumberUtils.createNumber("1l"); // Original body
3     // Test with a generalized input
4     String s = "" + ((char) n) + ((char) m) + "l";
5     String actOut = "";
6     try { actOut = "" + NumberUtils.createNumber(s).longValue(); }
7     catch (Exception e) { actOut = "Exception"; }
8     // Use Long.valueOf as a reference
9     String refOut = "";
10    try { refOut = "" + Long.valueOf(s); }
11    catch (Exception e) { refOut = "Exception"; }
12    Log.logOutIf(actOut.equals(refOut), () -> new String[] { actOut });
13 }
```

Figure 6: A generalized test for Lang58 (RI)

3.2.4 RI (Reference Implementation): We use a reference implementation that performs the same functionality in a preservation condition. It is known that developers often write redundant implementations[2, 3, 16]. Consider Figure 6 where the createNumber method is tested. The functionality of this method is the same as the Long.valueOf method when the input string to createNumber ends with “l”. The preservation condition, actOut.equals(refOut), expresses the intention that behavior should be preserved after patch when in the original version, the output of the method under test (actOut) is identical with the output of the reference implementation (refOut).

3.3 An interesting example:

Let's imagine a case where an APR tool generates a list of plausible fixes and the user has to select the correct one from the list. Assume the list includes both an incorrect patch (Figure 7(a)) and a correct patch (Figure 7(b)) for the same unstable version. It's important to bear in mind that the list is usually long. It's important to bear in mind that the list is usually long. For instance, more than 450 patches are generated by JAID [4] for that buggy program. To speed up the patch review process, the user can utilize a patch classification (PC) technique to filter out wrong patches before investigating the remaining patches. If the user utilizes one of the most up-to-date PC utilities, PATCH-SIM [39], the incorrect patch is not filtered out. PATCH- SIM, in fact, fails to reject 14 more incorrect patches for the same buggy version in our dataset. He might try ODS [43], a recent ML-based PC utility, when he's unsatisfied. Regrettably, he discovers that ODS is much more dissatisfying because it rejects the correct patch.

```
1    double ret;
2    double d = getDenominatorDegreesOfFreedom();
3  - ret = d / (d - 2.0);
4  + ret = d / (d + 2.0);
```

(a) An incorrect patch for Math95

```
1  - double ret;
2  + double ret = 1.0;
3    double d = getDenominatorDegreesOfFreedom();
4  + if (d > 2.0) {
5        ret = d / (d - 2.0);
6  + }
```

(b) A correct patch for Math95.

```
1  public void testSmallDegreesOfFreedom() {
2    FDistributionImpl fd =
3        new FDistributionImpl(1.0, 1.0);
4    double p = fd.cumulativeProbability(0.975);
5    double x = fd.inverseCumulativeProbability(p);
6    assertEquals(/* expected output */ 0.975, x,
7                 /* delta */ 1.0e-5);
8  }
```

(c) Developer-written failing test for Math95.

```
1  public void testSmallDegreesOfFreedom(double d1,
2    double d2, double d3) {
3    FDistributionImpl fd =
4        new FDistributionImpl(d1, d2);
5    double p = fd.cumulativeProbability(d3);
6    double x = fd.inverseCumulativeProbability(p);
7    // Which expression should be used in the following blank
8    // to express the correct output for a given random input?
9    assertEquals(/* expected output */ _____, x,
10                /* delta */ 1.0e-5);
11 }
```

(d) An incomplete parameterized test of (c).

Fig. 7. Motivating example.

3.4 Fuzzing:

Algorithm 1 Our differential fuzzing algorithm

Input: a buggy version p and its patched version p'
Input: a parameterized test with a preservation condition φ
Input: a set of patch location L
Output: witness value w ▷ $p(w) \neq p'(w) \wedge \varphi(w, p(w))$ if $w \neq \perp$

```
1:  w ← ⊥
2:  S ← ∅                                          ▷ a set of interesting input
3:  totalCov, totalCov ← ∅, ∅                      ▷ total coverage of p and p'
4:  rangeFixed ← false; noProgress ← 0
5:  repeat
6:      if S = ∅ then
7:          if noProgress >= T then
8:              WIDENRANGE( ); noProgress ← 0
9:          end if
10:         S ← r                                  ▷ r: random values within in the range
11:     end if
12:     s ← CHOOSENEXT(S)
13:     for i from 1 to ENERGY(s) do
14:         s† ← MUTATE(s, S)
15:         // Run the original version p
            ▷ o: outputs, cov: coverage, σ: program states
16:         o, cov, σ ← RUN(p, s†)
17:         if φ(s†, o) then
18:             noProgress ← 0; rangeFixed ← true
19:             // Run the patched version p'
                ▷ σ': dumped program state
20:             o', cov', σ' ← RUN(p', s†)
21:             if o ≠ o' then
22:                 w ← s†
23:                 return w                       ▷ return a found witness input
24:             end if
25:         else
26:             noProgress++; cov' ← ⊥
27:         end if
28:         if SHOULDSAVE(s†) then
29:             S ← S ∪ {s†}
30:         end if
31:     end for
32: until timeout reached
```

The fuzzing tool that is used for Poracle is JQF [28], a coverage-guided fuzzer for Java. However, we

extended the functionality of the tool so as to work with our tool. Here are the features we have added to JQF:

1. We provided specification APIs (logOutIf and ignoreOutOfOrg) that can help to write preservation conditions into the augmented tests in a more user-friendly way.
2. Our fuzzer runs the patched version only if there is an input that satisfies the given preservation condition. Otherwise, pre-patched version is run for the next random input. By doing so, we make sure that no resources (such as CPU) are wasted for uninteresting inputs.

Algorithm 1 shows our differential fuzzing algorithm which has the following novel features. First, preservation conditions are considered during the fuzzing process. If a random input does not satisfy the given preservation condition in the original version, the execution of the patched version is skipped, and a new random input is tried with the original version directly. Notice that in Algorithm 1, the run for the patched version (line 20) only takes place when the preservation condition is satisfied (line 17). Second, when Poracle chooses a random input in the range of $[c - \delta, c + \delta]$ where c is the original constant replaced by a parameter, δ is chosen adaptively. In general, the fuzzing space grows as a larger range is used, which can decrease the efficiency of fuzzing. Poracle gradually widens the range until a random input in the range satisfies a given preservation condition. In line 8, we widen the range if the given preservation condition is not satisfied for T consecutive times.1 In cases where a fixed range should be used for a certain parameter, we allow developers to express that intention, and adaptive widening is disabled.

Lastly, Poracle uses a novel metric to guide the fuzzing process. An earlier work such as AFLGo [1] uses the distance to the target to guide a directed fuzzing process. However, we found that when performing fuzzing with a unit test, a patched program location is usually easily reached. What is more challenging is to propagate state changes made in the patched location toward the end of the execution path. To achieve this, we propose a novel state-based metric in the following.

When a patch location is reached while executing a patched version, we extract the stack trace $m_1, m_2, \ldots, m_k$ where m_1 is the method to which the reached patch location belongs, m_k is the top-level method, and m_i is the caller of m_{i-1}. At each exit point of m_i, we extract a program state σ_i and σ'_i from a buggy version and its patched version, respectively. Note that a program state consists of a mapping between variables and their values. For each corresponding σ_i and σ'_i, we compute a state difference sd_i between them as their L_1 distance: $|| \sigma_i - \sigma'_i ||_1$. In the end, we obtain a sequence of state

differences, $sd_1, \ldots, sd_k$. When the state changes are not propagated up to the top-level method m_k, its state difference sd_k is 0. Intuitively, we want to propagate a non-zero distance observed in sd_1 up to m_k.

To realize our intention, we customize two key functions of grey-box fuzzing, that is, *Energy* and *ShouldSave*. The former determines how many consecutive times a chosen input is used to generate random input to the target program, and a more promising input should receive higher energy. Meanwhile, the *ShouldSave* function decides whether to keep the current input for later use, and thus useless input should not be saved. Our *Energy* function returns the value of the following formula. A similar formula was used in AFLGo [1].

$$MaxMutations \times energy(s_k, t)$$

where s_k and t represent the k-th input in input corpus S and the elapsed time of fuzzing, respectively. Function $energy(s_k, t)$ expresses how interesting s_k is at a given elapsed time t, and is defined as follows. Note that a return value of $energy(s_k, t)$ is in range between [0, 1].

$$\text{energy}(sk, t) = \frac{k}{|S|} \cdot (1 - T(t)) + 0.5 \cdot T(t)$$

where T (t) returns the temperature at time t and is defined as follows:$T(t) = 20^{-\frac{t}{tx}}$ Essentially, the formula returns a larger value as k increases at a given time t. To give higher energy to an input that propagates program changes further toward the end of the execution, we first sort the inputs in the corpus based on the state difference defined earlier. Given two sequences of state differences, $sd_1, \ldots, sd_k$ and $sd'_1, \ldots, sd'_k$, we compare each sdi and sd′ i starting from sdk . Note that the two sequences have the same length, since we extract the stack trace of the method containing the reached patched line, and the same input is used for the two versions. Currently, we do not consider cases where patched lines are scattered across multiple methods. As soon as a difference between sd_i and sd'_i for some i is found, we decide the order, ignoring remaining sub-sequences $sd_1, \ldots, sd_{i-1}$ and sd′ 1 , . . . , sd'_{i-1} . For example, given the following two sequences of state differences, [3, 2, 0] and [5, 0, 0], we consider that [3, 2, 0] > [5, 0, 0]. Although the first element is larger in the second sequence, the first sequence shows that state changes are propagated further than in the second sequence. Meanwhile, we

define the ShouldSave function in a way that the current mutated input $s^{\dagger}$ is saved in the following cases. (1) $s^{\dagger}$ propagates program changes further than the existing inputs in corpus S, (2) a new branch is covered in the patched version, and (3) the patched version is not executed since the preservation condition is not satisfied and a new branch is covered in the original version. Even when none of the above three conditions are satisfied, we also sometimes allow an input to be saved at a random rate using temperature function T (t) shown earlier. Note that T (t) decreases as elapsed time t increases, and the rate of random saving also proportionally decreases as t increases.

CHAPTER 4. Evaluation:

We evaluate our specification-based patch-validation approach from these three point of views.
(1) Performance: How does our technique carry out as compared to the latest approaches? Note that we use a black-box technique in which internal program state variations among patched and pre-patched variations aren't always propagated till the end. Here we want to see, how much of an impact does this have on patch validation's overall performance?
(2) Cost reduction: The significant cost of manually validating a large number of plausible patches motivates our study. Here we are looking for, how much cost reduction can be obtained by filtering out incorrect patches, when our methodology is applied to an APR tool that produces a ranked list of plausible patches.
(3) Suitability: In our approach, the users have to generalize an existing test by using a preservation condition. We want to check the difficulty level of writing this kind of condition. To evaluate this, we will compare the difficulty of writing preservation conditions with writing assertions in the conventional way.

4.1 Assessing Performance

At first we took the PATCH-SIM dataset (a de facto standard dataset) which has 139 patches. These 139 patches were prepared based on more than 70 buggy versions in the Defects4J benchmark[17]. For those versions, generalized tests were written. In total, there are 458 patches in our dataset. The PATCH-SIM dataset is used in the first experiment to ensure a proper comparison. Then we collect some more patches from [18]. From there we get 350 more patches for our 77 generalized or augmented tests. So now we have more than 450 patches (458 in total). To obtain those we had to delete some duplicate patches(31 in total). Here, duplicate patch means that, those patches were available in both datasets.

Table 1: Dataset used in our experiment.

Project	jGenprog	Genprog-A	jKali	Kali-A	Nopol2015	Nopol2017	ACS	HDRepair	ARJA	AVATAR	SimFix	DynaMoth	FixMiner	TBar	KPar	RSRepair-A	Total (Generated)	Developer Patches
Chart	6+0	0+0	6+0	0+0	6+0	6+0	0+2	0+0	0+0	0+0	0+0	0+0	0+0	0+0	0+0	0+0	24+2	13
Lang	0+0	0+0	0+0	0+0	5+2	10+2	2+3	0+1	0+1	4+0	0+4	1+2	1+0	5+2	4+1	0+1	32+19	10
Math	10+4	11+2	9+1	14+2	14+1	42+1	13+31	5+2	16+6	14+4	19+14	17+2	17+7	17+10	15+7	17+4	250+98	45
Time	2+0	0+0	2+0	0+0	1+0	9+0	0+3	0+1	0+0	1+2	1+1	1+0	1+2	1+2	1+2	0+0	20+13	9
Total	18+4	11+2	17+1	14+2	26+3	67+3	15+39	5+4	16+7	19+6	20+19	19+4	19+9	23+14	20+10	17+5	326+132	77

x+y denotes x patches labeled incorrect and y patches labeled correct.

In table 1 we can see the dataset that we used in our experiment. Here x represents those patches which are labeled incorrect and y indicates the patches labeled correct. In the dataset we have patches for 77 bug versions from Chart, Lang, Math and Time projects and PATCH-SIM also included these bugs in their experiment.

Project	UE	CC	EA	RI
Chart	8	4	1	0
Lang	0	2	4	6
Math	18	14	7	6
Time	3	0	4	0
Total	29	20	16	12

Table 2: The distribution of preservation invariant patterns.

4.1.1 Experimental Settings

We run our differential fuzzer with these generalized tests for maximum 10 minutes and the tests we generalized beforehand were used as fuzz drivers. All of our experiments are carried out on an Intel Xeon Gold processor with 128GB of RAM. PATCH-SIM, Opad, and two machine learning-based approaches are some of the best ways to classify patches. To see how well Poracle works, we compare it to each of them. We run PATCH-SIM with a 35-minute timeout, and it works well for us. PATCH-

SIM is said to take up to 30 minutes to make in its original work [39]. Randoop [27] is used inside PATCH-SIM, and we use a 3-minute time-out for Randoop which is same in the original experiment of PATCH-SIM. This is how we simulate Opad in Java: If a patched program throws an exception that isn't in the original version, we call the patch wrong. In the prior study [39], the same method was applied. Opad and Poracle are compared using the same fuzzer and timeout (i.e., 10 mins) since we believe this is the most fair way to compare the two tools.

Poracle is also compared to two machine learning-based techniques [33, 43]. Following the procedures outlined in [33], the patch classification process is carried out by first converting the patch code into an embedding vector using an embedding approach (such as BERT [7]), and then feeding the embedding vector to an ML classifier (such as logistic regression). In [33], it is demonstrated that the combination of BERT and logistic regression (referred to as BERT-LR in this research) performs the best, and we employ this combination in our comparison analysis. While the tool used in [33] isn't publicly available, the embedding vectors for the patches in their dataset, as well as the classification script, are, and we can use these to get BERT-LR results for 286 patches that exist in both our and their datasets. Using 332 patches in common between our dataset and theirs, we compare with another ML-based technique, ODS [43]. ODS uses hand-crafted code attributes to train a classifier.

4.1.2 Criteria of Assessment:

We utilize the following standard measures for precision and recall:

$$\text{Precision} = \frac{\textit{the number of rightly rejected patches}}{\textit{the total number of rejected patches}}$$

$$\text{Recall} = \frac{\textit{the number of rightly rejected patches}}{\textit{incorrect patches in total}}$$

Table 3: Comparison between Our tool, PATCH-SIM , and Opad

Project	Patches		Rejected Rightly			Wrongly Rejected		
	Incorrect	Correct	Ours	PATCH-SIM	OPAD	Ours	PATCH-SIM	OPAD
Chart	24 / 24	2/2	17 / 17	14/14	2/2	0/0	0/0	1/1
Lang	11 / 32	4 / 19	9 / 19	6/11	1/1	0/0	0/6	1/1
Math	64 / 250	19 / 98	42 / 148	33/66	17/40	0/2	0/3	4/19
Time	13 / 20	2 / 13	10 / 13	9/10	7/8	0/0	0/0	0/0
Total	112/326	27/132	78/197	62/101	27/51	0/2	0/9	6/21

We use 139 (=112 incorrect + 27 correct) patches obtained from the PATCH-SIM dataset [8] and the extended dataset [19] which was shown in the previous table (table #3). In notation x/y , x and y represent the data for the first dataset and the extended dataset, respectively. In each row, the best results are highlighted.

Poracle's performance is compared to those of PATCH-SIM and Opad in Table 3. The "Patches" column in the table displays the number of incorrect and correct patches in our dataset. The data for the PATCH-SIM dataset and the extended dataset (Table 1) are represented by x and y, respectively. This difference is drawn to present the PATCH-SIM results for the PATCH-SIM dataset as published by its authors [39]. In the meantime, the "Rightly Rejected" ("Wrongly Rejected") column displays the number of incorrect patches that have been correctly (incorrectly) recognized as incorrect. The results for the first and extended datasets are represented by x and y, respectively, in the notation x/y. All results, with the exception of x of PATCH- SIM, were acquired by us. The best outcomes are highlighted in each row of the table. Ours outperforms PATCH-SIM and Opad in each of the four projects. Poracle has a recall rate of 60% (197/326), which is over twice that of PATCH-SIM (31%), and four times that of Opad (16 percent). Ours also has a precision of 99 percent (197/(197 + 2)), which is higher than PATCH-SIM (92 percent) and Opad (71 percent). False positives are more common in Opad than in the other instruments. This is because in Java programs, throwing an exception isn't always a bug, and some tests expect an exception to be thrown. It's difficult to know whether an exception is expected or not without a user definition. The performance of Opad, for which we use the same fuzzer as Poracle, is the worst, emphasizing the importance of incorporating the specification.

Table 4: Comparison between Poracle and BERT-LR

Project	Patches		Rightly Rejected		Wrongly Rejected	
	Incorrect	Correct	Poracle	BERT-LR	Poracle	BERT-LR
Chart	24 / 24	2 / 2	17 / 17	16 / 16	0 / 0	0 / 0
Lang	11 / 24	4 / 8	9 / 14	1 / 6	0 /0	0 / 1
Math	64 / 167	19 / 42	42 / 105	23 / 73	0 / 1	0 / 1
Time	13 / 16	2 / 3	10 / 11	3 / 4	0 / 0	0 / 0
Total	112 / 231	27 / 55	78 / 147	43 / 99	0 / 1	0 / 2

We consider two separate datasets for a proper comparison: (1) 139 (=112+27) patches used in the PATCH-SIM study [39] (the same as in Table 3) and (2) 286 (=231+55) patches, which are the intersection of the dataset available in [33] and our extended dataset (Table 1).

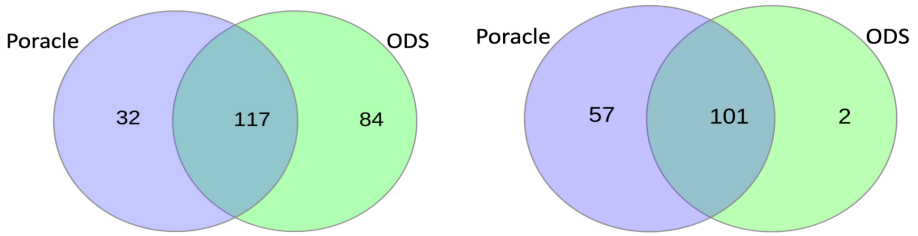
Poracle's performance is compared to that of BERT-LR [33] in Table 4. We continue to use the x/y notation. Poracle also outperforms BERT-LR, according to our findings. Poracle's recall (64 percent) is roughly 1.5 times higher than BERT-LR's (43 percent), while precision is high both in Poracle (99 percent) and BERT-LR (43 percent).

Table 5: Comparison between Poracle and ODS

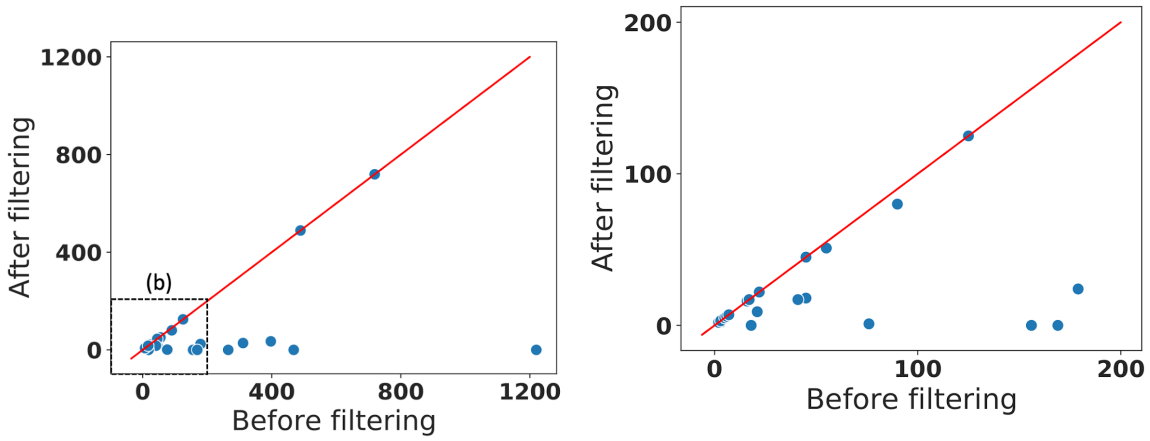
Project	Patches		Rightly Rejected		Wrongly Rejected	
	Incorrect	Correct	Poracle	ODS	Poracle	ODS
Chart	23 / 23	2 / 2	16 / 16	13 / 13	0 / 0	0 / 0
Lang	10 / 26	3 / 10	8 / 15	9 / 25	0 /0	0 / 7
Math	60 / 177	19 / 64	41 / 108	33 / 149	0 / 1	3 / 15
Time	13 / 16	2 / 7	10 / 11	11 / 14	0 / 0	1 / 6
Total	106 / 242	26 / 83	75 / 149	66 / 201	0 / 1	4 / 28

Poracle and ODS are compared in Table 5. In the extended dataset, ODS has a greater recall than Poracle, however Poracle outperforms ODS in the PATCH-SIM dataset. Poracle has a precision of 99 percent, however ODS rejects the most accurate patches (28) of all the approaches studied. ODS

rejects fixes more forcefully in general, regardless of whether they are correct or erroneous. Figure 7 clearly depicts this trend. ODS correctly rejects 52 more incorrect patches than Poracle, as indicated in the diagram on the left. The right diagram 7, on the other hand, demonstrates Poracle's superiority over ODS by accepting 55 more correct patches.



(a) Rightly rejected patches **(b) Rightly accepted patches**



(a) Overall results. **(b) Results for the patches whose original rankings are below 200.**

Figure 8: The number of patches to review before (X-axis) and after (Y-axis) applying our approach.

Figure 8 shows our results. In most cases (20 out of 28), (JAID was successful in generating a patch list for the 28 buggy versions in our dataset.) the number of patches to review decreases. For example, in Chart9, a correct patch originally ranked at 45th is ranked up to 18th after applying our approach. On average, 108 patches are reduced from the number of patches to review. Our results suggest that the manual validation effort can be reduced using our method.

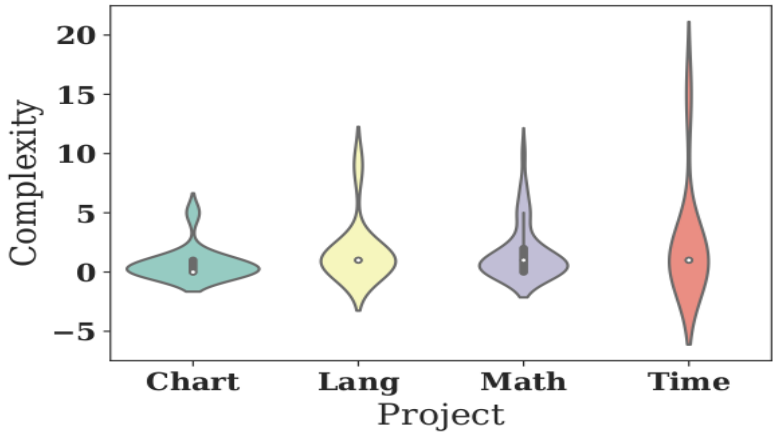


Figure 9: The complexities of the preservation conditions we wrote.

Assessing Usability:

We hypothesize that developers with domain-specific knowledge of a patched program can easily write a preservation condition. In this first study on preservation conditions, we assess the usability in the following two ways. First, we estimate the complexities of the preservation conditions we wrote for our experiments, and secondly, we conduct a user study with students.

Complexities. We estimate the complexities of the preservation conditions we wrote by counting the number of operators (e.g.,>, >=, && and ||) used in a preservation condition. As shown in Figure 8, we found that the preservation conditions we wrote are usually quite simple requiring on average only 1.58 operators.

User Study Setup. Our participants are 66 junior/senior students enrolled in a Software Engineering course opened in a university (names are anonymized). 34% of the participants have 4-6 years of programming experience, while 55.4% and 10.7% of the participants have 1-3 and 7-9 years of programming experience, respectively. To motivate the participants, grade compensation was provided. Our survey questions (available in the supplementary material) are designed based on four bugs of Defects4J (Math 28, 105, 73, and Lang 58). For each bug, we provided 10 patches, in which one of them is a correct patch and we asked participants to find out the correct patch. For our method, we showed the participants a failing test and its incomplete generalized version for each bug to fill in to run our tool to write down the patch that got accepted by our tool and for conventional method, we asked participants to manually investigate given patches in order to find out the correct one. We provided the participants with all necessary information to write correct answers, such as API documents.

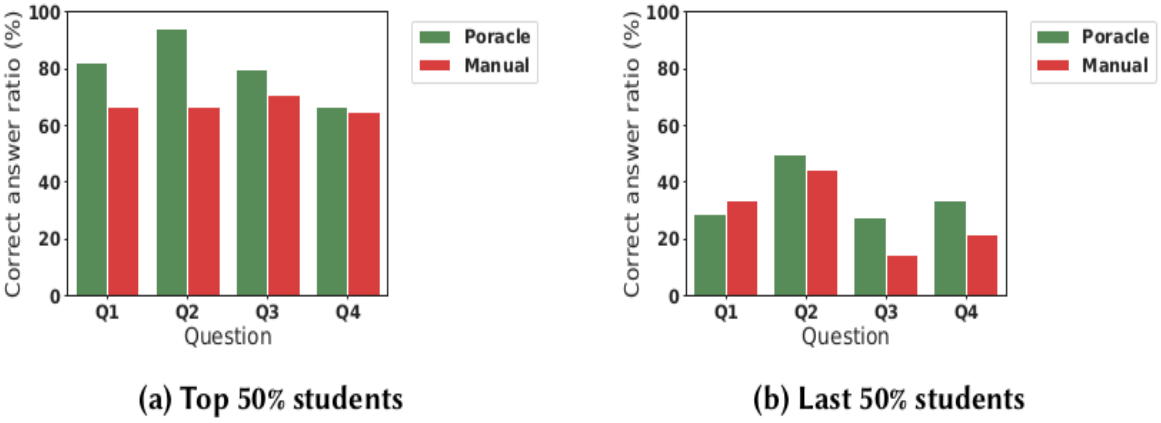


Fig. 10. The percentage of correct answers per question for methods.

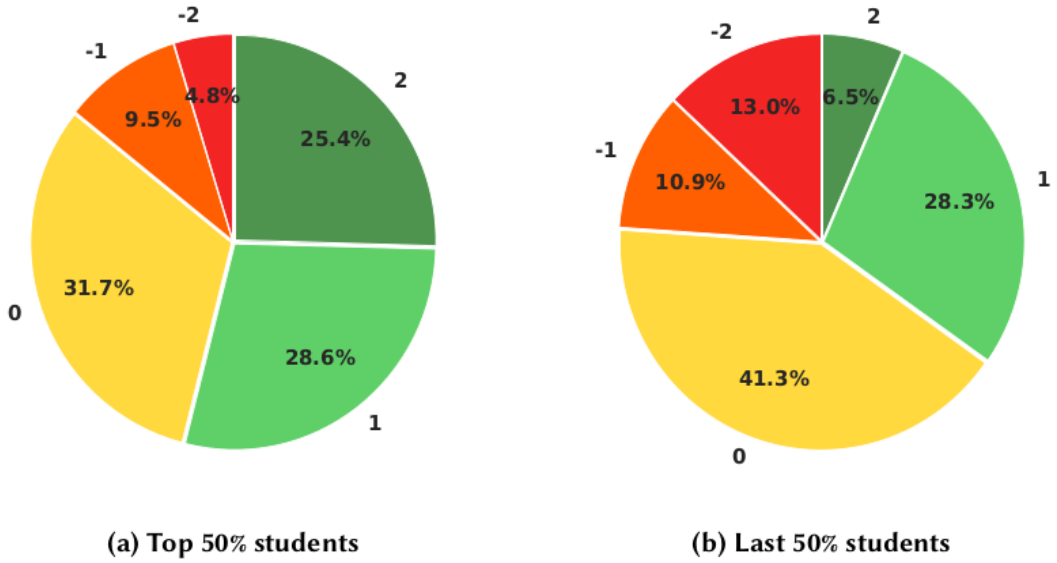


Fig. 11. The experience of the participants for Poracle.

We divided the 66 participants into two groups, and different questions were assigned between these groups. We assigned each group four questions with two questions for each method. For each pair of questions prepared for the same bug, we distributed the two questions into two different groups to prevent the result of one question from being affected by another question. Before conducting the survey, we showed a separate example of generalizing an existing test with our method.

When assessing the answers of the participants, we gave either 1 point (if an answer is correct), 0 points (if an answer is completely wrong).

User Study Results. In figure 10, we can see that the students tend to give correct answers using our tool more frequently than the manual method across almost all the questions. Figure 11 shows the

distribution of the number of students who answered correctly per question as a count plot. Overall, slightly more students per question were able to write correct preservation conditions and hence found the correct patches accordingly. The reason for the small gap is explained in the next section. Figure 12 shows the distribution of the number of students who expressed their overall experience about both methods as a pie chart and we can see that, more students were positive about their experience with our tool compared to the manual method. At the end of the survey we asked students about their preferred method and the majority of students (87.3%) preferred our tool over the manual method (12.7%), as shown in figure 13(a). In figure 13(b) we can see that, most of the participants preferred our tool over ML based fully automatic approaches (one of the reasons behind this is that, ML-based approaches do not provide evidence why a patch is correct or incorrect).

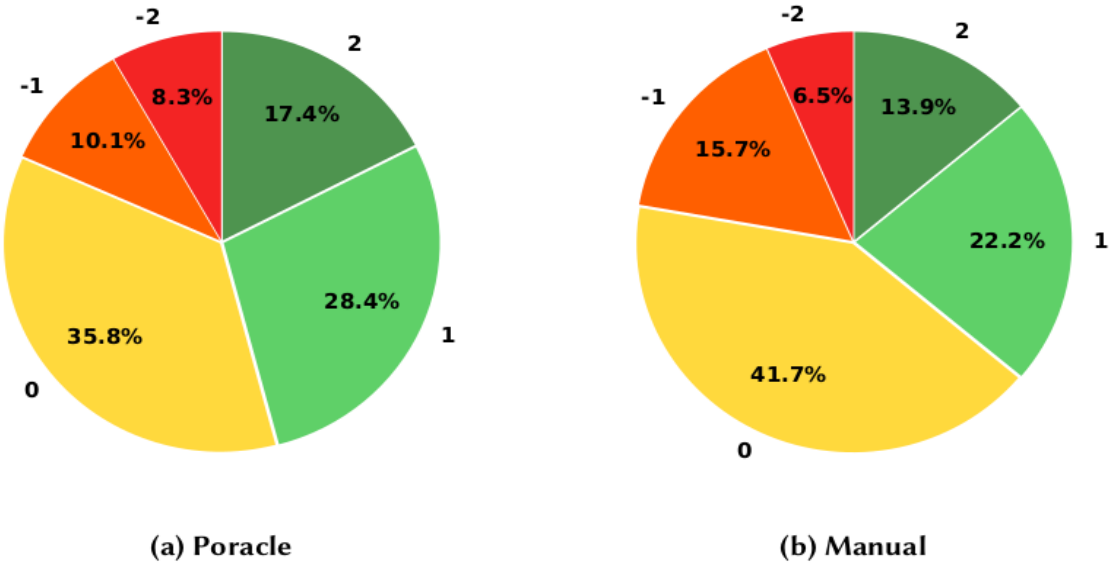


Fig. 12. The experience of the participants.

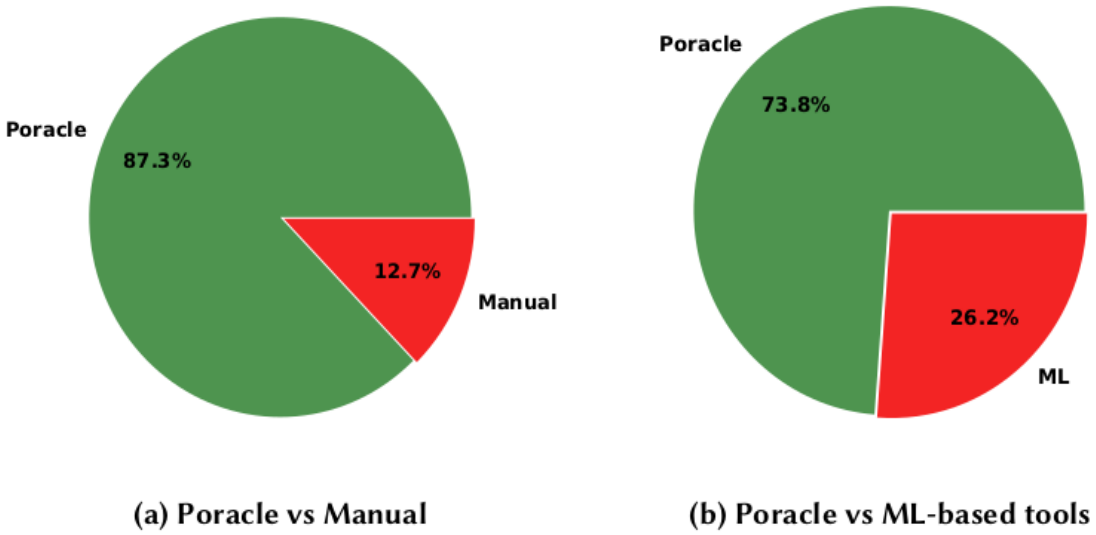


Fig. 13. The preferences of the participants.

Conclusion:

This work shows a lightweight semi-automatic methodology through which an existing failing test is general- ized, and then incorrect patches are filtered out via differential fuzzing. Our experimental results show that the benefit of using a generalized test is high; a large portion of incorrect patches are filtered out while nearly all correct patches are retained. These unique features of our approach not shared with the existing approaches — i.e., high recall and near- perfect precision — effectively reduce the manual cost of patch review. This benefit is obtained only at the cost of generalizing a single test. To the best of our knowledge, this is the first work that provides a concrete evidence that having developers more actively participate in the process of APR can be paid off. Furthermore, our user-study results show that the users prefer to use this tool than using the manual method.

Future work:

Automation of generalizing tests: The generalization of tests can be partially automated. More specifically, the randomization of parameters and modification of test can be done automatically using a script. The only thing that should be done by users is, writing the preservation invariant.

Once we acquire domain-specific knowledge about the bugs, we found that writing preservation invariant is usually simple. This is owing in part to the fact that preservation conditions are often straightforward, requiring only a few operators. Figure 8 depicts the number of operators (such as >, >=, &&, and ||) employed in preservation conditions across four different subjects in our dataset. The average complexity of the preservation invariant we defined (the average number of used operators) is only 1.58. Even the process of writing a condition can be partly auto- mated in a similar way to refactoring incorporated into IDEs such as IntelliJ [14] and Visual Studio Code [22]. For example, when the user observes an unexpected exception from a test, automatic transformation of an existing test (for example the generalized version (e.g., Figure 3)) can be performed straightforwardly. For the remaining less common patterns, more user input is necessary.

References

[1] Michael Buckland and Fredric Gey. 1994. The relationship between recall and precision. Journal of the American society for information science 45, 1 (1994), 12–19.

[2] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolo Perino, and Mauro Pezze. 2013. Automatic recovery from runtime failures. In 2013 35th International Conference on Software Engineering (ICSE). IEEE, 782–791.

[3] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. 2010. Automatic workarounds for web applications. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. 237–246.

[4] Liushan Chen, Yu Pei, and Carlo Alberto Furia. 2020. Contract-based program repair without the contracts: An extended study. IEEE Transactions on Software Engineering (2020).

[5] Tsong Yueh Chen, DH Huang, TH Tse, and Zhi Quan Zhou. 2004. Case studies on the selection of useful relations in metamorphic testing. In Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004). Citeseer, 569–583.

[6] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In International Conference on Functional Programming (ICFP). 268–279.

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

[8] Thomas Durieux and Martin Monperrus. 2016. IntroClassJava: A benchmark of 297 small and buggy Java programs. (2016).

[9] The Apache Software Foundation. 2021. The API of inverseCumulativeProbability.https://commons.apache.org/proper/commons-math/javadocs/api3.6/org/apache/commons/math3/distribution/NormalDistribution.html#inverseCumulativeProbability(double). Accessed August 30, 2021.

[10] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. 8–18.

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

[12] Chris Hawblitzel, Ming Kawaguchi, Shuvendu K Lahiri, and Henrique Rebêlo. 2013. Towards modularly comparing programs using automated theorem provers. In International Conference on Automated Deduction. Springer, 282–299.

[13] Paul Holser. 2014. junit-quickcheck: Property-based testing, JUnit-style. https://pholser.github.io/junit-quickcheck/. Accessed December 28, 2021.

[14] JetBrains. 2000. IntelliJ IDEA. https://www.jetbrains.com/idea/. Accessed August 30, 2021.

[15] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In ISSTA. ACM, 298–309.

[16] Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equiv- alent code fragments via random testing. In Proceedings of the eighteenth interna- tional symposium on Software testing and analysis. 81–92.

[17] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of ex- isting faults to enable controlled testing studies for Java programs. In International Symposium on Software Testing and Analysis (ISSTA). 437–440.

[18] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In Proceedings of the 2013 International Conference on Software Engineering (San Francisco, CA, USA) (ICSE '13). IEEE Press, Piscataway, NJ, USA, 802–811. http://dl.acm.org/citation. cfm?id=2486788.2486893

[19] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In International Conference on Software Engineering (ICSE). 615–627.

[20] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In POPL. 298–312.

[21] Fan Long and Martin C. Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In ICSE. 702–713.

[22] Microsoft. 2021. Visual Studio Code. https://code.visualstudio.com/. Accessed August 30, 2021.

[23] Amirfarhad Nilizadeh, Gary T Leavens, Xuan-Bach D Le, Corina S Păsăreanu, and David R Cok. 2021. Exploring true test overfitting in dynamic automated program repair using formal methods. In IEEE Conference on Software Testing, Verification and Validation (ICST). IEEE, 229–240.

[24] Yannic Noller, Corina S Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. 2020. HyDiff: Hybrid differential software analysis. In International Conference on Software Engineering (ICSE). IEEE, 1273–1285.

[27] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007.

Feedback-directed random test generation. In International Conference on Software Engineering (ICSE'07). IEEE, 75–84.

[28] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: coverage-guided property-based testing in Java. In ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). 398–401.

[29] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a doubt: testing for divergences between software versions. In International Conference on Software Engineering (ICSE). 1181–1192.

[30] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. 2017. Nezha: Efficient domain-independent differential testing. In IEEE Symposium on security and privacy (SP). IEEE, 615–632.

[31] 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 ESEC/FSE. 532–543.

[32] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In International Symposium on Foundations of Software Engineering (FSE). 727–738.

[33] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F Bissyandé. 2020. Evaluating representation learning of code changes for predicting patch correctness in program repair. In International Conference on Automated Software Engineering (ASE). IEEE, 981–992.

[34] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests. ACM SIGSOFT Software Engineering Notes 30, 5 (2005), 253–262.

[35] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated patch correctness assessment: How far are we?. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. 968–980.

[36] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In ICSE. ACM, 1–11.

[37] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: Balancing Edit Expressiveness and Search Effectiveness in Au- tomated Program Repair. In Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE). 354–366.

[38] Qi Xin and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In International Symposium on Software Testing and Analysis (ISSTA). 226–236.

[39] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In International Conference on Software Engineering (ICSE). 789–799.

[40] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In ICSE. 416–426.

[41] Bo Yang and Jinqiu Yang. 2020. Exploring the differences between plausible and correct patches at fine-grained level. In 2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF). IEEE, 1–8.

[42] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In Joint Meeting on Foundations of Software Engineering (FSE). 831–841.

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

[44] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the QuixBugs benchmark. Journal of Systems and Software 171 (2021), 110825.

[45] Jooyong Yi, Dawei Qi, Shin Hwei Tan, and Abhik Roychoudhury. 2015. Software change contracts. ACM Transactions on Software Engineering and Methodology (TOSEM) 24, 3 (2015), 1–43.

Acknowledgement

I would like to thank my supervisor, Professor Jooyong Yi, for his patience, guidance, and support. I have benefited greatly from your wealth of knowledge and meticulous editing. I am extremely grateful that you took me on as a student and continued to have faith in me over the years. I also thank Professor Dongsun Kim (KNU) for generously taking time out of his schedules to participate in this research and make this project possible.

Thank you to my committee members, Professor Mijung Kim and Professor Yuseok Jeon. Their encouraging words and thoughtful, detailed feedback have been very important to me. Their constructive criticisms had helped my work become well-polished and up to the required standard.

I also had the pleasure of working with Elkhan Ismayilzada, who helped me a lot in this project. I have learned many things from him while working on this project.

Finally, I would like to thank my family and friends, whose constant love and support keep me motivated and confident. My accomplishments and success are because they believed in me. Last but not the least, I thank the students who participated in the user study and gave their valuable feedback.