Poracle: Testing Patches under Preservation Conditions to Combat the Overfitting Problem of Program Repair

ELKHAN ISMAYILZADA and MD MAZBA UR RAHMAN, UNIST, South Korea
DONGSUN KIM, Kyungpook National University, South Korea
JOOYONG YI, UNIST, South Korea

To date, the users of test-driven program repair tools suffer from the overfitting problem; a generated patch may pass all available tests without being correct. In the existing work, users are treated as merely passive consumers of the tests. However, what if they are willing to modify the test to better assess the patches obtained from a repair tool? In this work, we propose a novel semi-automatic patch-classification methodology named Poracle. Our key contributions are three-fold. First, we design a novel *lightweight* specification method that reuses the existing test. Specifically, the users extend the existing failing test with a *preservation condition*—the condition under which the patched and pre-patched versions should produce the same output. Second, we develop a fuzzer that performs differential fuzzing with a test containing a preservation condition. Once we find an input that satisfies a specified preservation condition but produces different outputs between the patched and pre-patched versions, we classify the patch as incorrect with high confidence. We show that our approach is more effective than the four state-of-the-art patch classification approaches. Last, we show through a user study that the users find our semi-automatic patch assessment method more effective and preferable than the manual assessment.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Automatic programming;**

Additional Key Words and Phrases: Automated program repair, overfitting problem, patch validation, patch classification, preservation condition

ACM Reference format:
Elkhan Ismayilzada, Md Mazba Ur Rahman, Dongsun Kim, and Jooyong Yi. 2023. Poracle: Testing Patches under Preservation Conditions to Combat the Overfitting Problem of Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 44 (December 2023), 39 pages.
https://doi.org/10.1145/3625293

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1A2C1009819, No. 2021R1A5A1021944, No. 2021R1I1A3048013) and the Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2021-0-01001).
Authors' addresses: E. Ismayilzada, M. M. Ur Rahman, and J. Yi (Corresponding author), Department of Computer Science and Engineering, Ulsan National Institute of Science and Technology, 50 UNIST-gil, Ulsan 44919 South Korea; e-mails: {elkhan, mazbaur, jooyong}@unist.ac.kr; D. Kim, Kyungpook National University, Daegu, South Korea; e-mail: darkrsw@knu.ac.kr.

1049-331X/2023/12-ART44 $15.00
https://doi.org/10.1145/3625293

1 INTRODUCTION

Automated program repair (APR) techniques [18, 24, 26, 32, 33] have been developed extensively over the past decade. In particular, the ***generate-and-validate* (G&V)** approaches have gained wide popularity. A patch candidate P_c generated in the first step is validated in the second step (see Figure 1), typically with a user-given test suite. If P_c passes all tests, then many APR systems stop the repair process and show P_c to the user. However, there is no guarantee that P_c is a correct patch; P_c may merely pass all tests without being correct, which is often regarded as a *plausible but incorrect patch*. This problem is called an *overfitting problem* in the APR literature [36, 46, 51, 60].[1]

To overcome the overfitting problem, many recent APR systems generate a list of plausible (i.e., test-suite-passing) patches (instead of a single patch) so a correct patch existing in the patch space is not missed out [4, 11, 12, 22, 53, 54]. In this case, the users should *manually* look through a list of plausible patches to find the correct one. Because a test suite is only an under-constrained specification, hundreds and sometimes even thousands of plausible patches are generated [4, 11, 12, 29, 53], and the manual assessment step can take a long time. While patches can be reviewed[2] in the order of their rankings [4, 53], ranking algorithms are often imprecise, and a correct patch is not necessarily ranked high.

Existing work. Recently, several **patch classification (PC)** techniques [11, 49, 56, 59, 60] have been suggested to filter out plausible yet incorrect patches, as listed in the first column of Table 1 in Section 2. These PC techniques can be broadly classified into *score-based* approaches [49, 56, 60] and *evidence-based* approaches [11, 59]. The former approaches compute a score of a patch based on various information such as distributed representations of patches [49] and path spectra of patched and pre-patched (i.e., buggy) versions [56]. The computed score of a patch is compared with a chosen threshold to make a classification decision. Meanwhile, the latter approaches use differential testing [38, 41, 43] to detect behavioral differences between pre-patched and patched versions [11, 59]. If the output of the pre-patched version is correct, then the observed behavioral difference indicates a regression error. Since an incorrect patch is rejected only with concrete evidence of a regression error, *perfect precision* can be achieved (i.e., a patch classified as incorrect is indeed incorrect), provided that the discovered behavioral difference correctly identifies an error. This is the clear advantage of the evidence-based approach over the score-based approach.

Please note that we define **true positive (TP), false positive (FP), true negative (TN)**, and **false negative (FN)** as follows, considering the context of patch classification. TP refers to an incorrect patch being correctly classified as incorrect, while FP refers to a correct patch being erroneously classified as incorrect. However, TN and FN refer to a correct patch being correctly classified as correct and an incorrect patch being incorrectly classified as correct, respectively. Recall ($TP/(TP + FN)$) in our context measures how often incorrect patches are successfully filtered out, while precision ($TP/(TP + FP)$) measures how often a filtered-out patch is indeed incorrect. These definitions of recall and precision are consistent with previous work on patch classification [49, 51, 56, 60].

Both types of PC techniques have their limitations. For score-based approaches, it is difficult to choose a threshold that makes recall high (i.e., incorrect patches should be filtered out as much as possible) while keeping precision close to 100% (i.e., correct patches should not be filtered out) [1]. Meanwhile, evidence-based approaches suffer from low recall. This is because it is difficult to deter-

[1]Although the term "overfitting" was originated from machine learning vocabulary, an APR system usually does not use held-out tests. Typically, all available tests are provided to an APR system. In the APR literature, an *overrating* patch is often used as a synonym of an *incorrect* patch.

[2]We use "review" and "assess" in an interchangeable way.

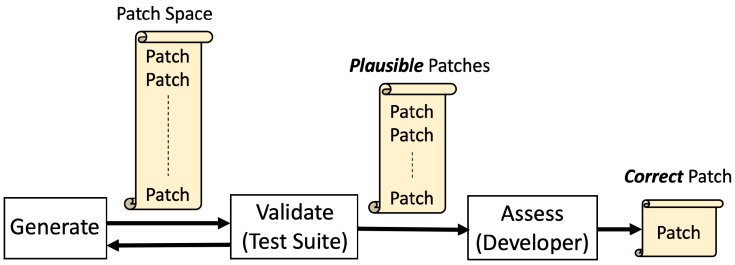


Fig. 1. Three steps used in recent *generate-and-validate* (G&V) APR tools generating a list of plausible patches, among which the developer should *manually* look for a correct patch.

mine whether a detected output difference between pre-patched and patched versions is indicative of a regression error, without knowing whether that difference is intended or not, and to avoid this issue, existing approaches [11, 45, 59] filter out only patches that result in program crashes.[3] As will be shown in this article, we overcome this limitation by leveraging a lightweight specification methodology.

Our viewpoint. The overfitting problem of APR occurs because 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 tests are prepared without considering a bug at hand. This results in a situation where the user of an APR system should rely on luck to obtain the correct patch. There is no wonder why APR systems suffer from the overfitting problem!

In this work, we pose the following question: *What if the users are willing to express the bug-fixing intention to better assess the obtained patches?* With that question in mind, we suggest a semi-automatic patch-classification methodology named Poracle that combines the evidence-based PC technique with a *lightweight* specification methodology. With Poracle, the user can specify the following kind of assertion in a test:

$$\forall x : \textit{whenever } \varphi(\vec{x}) \textit{ is satisfied, } p(\vec{x}) = p'(\vec{x}), \tag{1}$$

where p and p' refer to a buggy version and its patched version, respectively, and $\vec{x}$ represents the input to p and p', including global context such as timezone. We assume that p and p' are deterministic on $\vec{x}$ and do not consider flaky tests in this work, as commonly assumed in previous work on APR. Meanwhile, φ refers to our novel *preservation condition*—the condition under which program behavior should be preserved between p and p'. Once the user specifies a preservation condition, Poracle performs differential fuzzing to search for a value v violating condition (1); then, v is used as evidence to classify p' incorrect.

Comparison with formal specification. Incorrect patches generated from APR tools can also be filtered out using formal specifications, as shown by Nilizadeh et al. [37]; patches that violate the specification are considered incorrect. However, writing precise formal specifications is very challenging, as shown by Legunsen et al. [25]. According to their study, the developers often write imprecise specifications, which results in a large number of false alarms. To write a precise specification, the developers need to consider all possible cases and should come up with a precise oracle for each case. In contrast, writing a preservation condition is much simpler, since it only requires considering a specific context in which the bug appears.

[3]Vulnerabilities detected by sanitizers such as UBSan and ASan and violation of in-code assertions are considered crashes as well.

Contributions. The main contributions of this work are:

- **A novel patch classification methodology:** We propose a novel semi-automatic patch classification methodology by combining an evidence-based technique and a lightweight specification method with which a preservation condition can be specified. Please note that our method is not intended to replace the existing automatic patch classification techniques. Instead, we provide the developers who prefer to have control over the patch classification process with a concept (i.e., preservation condition) and tools they can use (i.e., APIs to express preservation conditions and a differential fuzzer).
- **Empirical findings:** We run Poracle over the 458 patches in our dataset after adding preservation conditions. Compared with four state-of-the-art PC techniques [49, 56, 59, 60], our approach substantially outperforms three of them [49, 56, 59] both in recall and precision. While ODS [60] shows better recall than Poracle, ODS shows the lowest precision among all tools, implying that ODS erroneously rejects correct patches most frequently. Considering the scarcity of correct patches [29] and the objective of program repair (i.e., finding a correct patch), rejecting a correct patch is the last thing we want. In contrast, Poracle shows 99% precision. Our experimental results show that many incorrect patches an APR tool generates can be filtered out *only with a snippet of information provided through a preservation condition* (note that a preservation condition only specifies what should be preserved but does not specify what changes should be made).
- **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:** The result of our user study conducted with 66 participants shows that the users find the correct patch more often when Poracle is provided than when manually finding the correct one. Most participants of our user study preferred the semi-automatic patch assessment using Poracle to the manual assessment.
- **Replication package:** We provide a replication package in the following URL:

  > *https://github.com/UNIST-LOFT/poracle.*

  Our package contains all generalized tests covering 458 patches and our custom fuzzer supporting our novel preservation conditions.

2 BACKGROUND AND RELATED WORK

2.1 Automated Program Repair (APR)

The goal of **automated program repair (APR)** is to automatically generate a patch that fixes a given bug. Over the past decade, diverse APR techniques have been developed. Some techniques [12, 24] randomly mutate the given buggy program under the guidance of a test suite *TS* with an aim to obtain a mutated program that passes all tests in *TS*. Other techniques [20, 26] use predefined templates to generate patches instead of random mutations. More recent techniques [18, 54] employ machine-learned models to generate patches. Meanwhile, semantics-based approaches [32, 33] first infer constraints to fix the bug and then generate a patch that satisfies the constraints.

The current challenges of APR include (1) how to fix more diverse bugs [44], (2) how to generate patches fast [22, 62], and (3) how to generate correct patches. This work focuses on the third issue.

Most current APR techniques cannot guarantee the correctness of generated patches due to the overfitting problem described below.

2.2 Overfitting Problem

One of the major challenges in APR is the overfitting problem [46] that occurs because many *incorrect* patches exist in the patch space; they are often *not filtered out* by a given test suite [29, 46]. To address this problem, many APR systems use one of the three techniques: (1) patch ranking algorithms, (2) score-based patch classification, and (3) evidence-based patch classification.

The idea of patch ranking is to identify patches that are more likely to be correct and place them at higher rankings than the others. For example, Prophet [28] ranks patch candidates based on a probabilistic model learned from existing patches. Other APR tools, such as ACS [57], CapGen [52], SimFix [16], and JAID [4], similarly perform patch ranking.

However, the ranking algorithms are often imprecise, and it is common that a correct patch is *not ranked first*. As the patch space increases, this problem tends to be exacerbated, and correct patches are often missed out [29]. Thus, many recent studies propose improved PC techniques [11, 47, 49, 51, 56, 59, 60]. The goal of the PC technique is to filter out incorrect patches while keeping correct patches.

Score-based PC techniques perform patch classification by computing the scores of the patches. In anti-patterns (common patterns of incorrect patches) [47], a simple binary scoring scheme is used; patches belonging to anti-patterns receive a low score and are rejected. Otherwise, patches are accepted. A more recent technique, PATCH-SIM [56], computes the path similarity between the execution paths before and after the patch. Various machine-learning-based classification techniques have also been developed, using hand-crafted features [60] or embedding techniques (e.g., Reference [49] and BATS [48]). These techniques compare the computed scores with a threshold to perform classification, and the threshold is typically chosen empirically (e.g., using training data). In general, it is *difficult to choose a threshold* that makes recall high while keeping precision close to 100% [1]. Also, these approaches do *not* provide a *semantic explanation* for the classification decision.

Evidence-based approaches [11, 59] do not have the limitations of the score-based approaches, since a patch is rejected only with concrete evidence of the error. Opad [59] uses a fuzzer to detect crashing patches. Fix2Fit [11] similarly uses a fuzzer to avoid generating crashing patches. However, since only crashing patches are detected, the recall of the existing evidence-based approaches is low.

Table 1 summarizes the existing PC techniques. Our technique inherits the advantage of evidence-based techniques—i.e., when an incorrect patch is filtered out, clear evidence for the rejection is provided to the user—and at the same time, lifts the limitation of the existing evidence-based techniques—i.e., low recall due to the fact that only crashing patches can be detected. By allowing the user to specify his or her intention for the patch, our method can also detect non-crashing incorrect patches, which leads to high recall, as will be shown with experiments. In the PC classification research, the main goal has been to achieve high recall without dismissing a correct patch [51]. In this work, we show that this goal can be achieved by taking only a snippet of information from the user.

2.3 Program Contracts

In the recent work of Nilizadeh et al. [37], program contracts are used to verify the correctness of APR-generated patches using formal verification techniques. Program contracts were also used to express and verify the user's change intention [13, 23, 63, 64]. Compared with these more formal

Table 1. Comparison between Patch Classification (PC) Techniques

PC Technique	Filtering Criterion	No False Positive	Requires Threshold Tuning	Detectable Error Type	Allows User Spec
Anti-patterns [47]	Anti-patterns	✗	✗	Anti-patterns	✗
PATCH-SIM [56]	Score	✗	✓	General	✗
Tian et al. [49]	Score	✗	✓	General	✗
ODS [60]	Score	✗	✓	General	✗
OPAD [59]	Evidence	✓*	✗	Crash	✗
FIX2FIT [11]	Evidence	✓*	✗	Crash	✗
Ours	Evidence	✓†	✗	General	✓

*: Under the assumption that the observed crash of the patched version indicates a regression error.
†: Under the assumption that the user-provided preservation condition is either precise or under-approximate (see Section 5.2.1).

```
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. 2. Motivating example.

approaches, our lightweight approach does not require a separate contract, and our preservation condition is *directly* added into an existing failing test, the input of most APR tools.

2.4 Patch Evaluation

Apart from PC techniques, there are also **patch evaluation (PE)** techniques [55, 58, 61, 65] where patch correctness is evaluated *based on correct versions in the benchmark*. Unlike these patch evaluation techniques, PC techniques like ours classify the correctness of a patch *without consulting correct versions*.

3 A MOTIVATING EXAMPLE

Consider a scenario where an APR tool returns a list of plausible patches, and the user finds a correct patch among them. Suppose that the list contains many incorrect patches, including the one shown in Figure 2(a) and a correct patch shown in Figure 2(b), all of which pass all available

tests. Note that the size of the list is often large. For example, JAID [4] generates 1,263 patches[4] for the example buggy version (Math95 in the Defects4J benchmark [19]). To expedite the patch review process, the user may want to first filter out incorrect patches using a PC technique before reviewing the remaining patches. If she uses PATCH-SIM [56], one of the state-of-the-art PC tools, then the example incorrect patch is failed to be filtered out. In fact, all (14) incorrect patches for the same buggy version available in our dataset are failed to be filtered out by PATCH-SIM. Being disappointed, she may try out a recent ML-based PC tool, ODS [60]. Unfortunately, she only finds that ODS is even more disappointing, since it filters out the correct patch!

This example illustrates the challenge of patch classification. The users would want to filter out incorrect patches as much as possible, but the last thing they would want is to discard correct patches, which are only *scarcely* available. Both PATCH-SIM [56] and ODS [60] use score-based approaches, making it challenging to distinguish between incorrect and correct patches without discarding the latter. They compute a score for a given patch and make a classification decision by comparing the obtained score with a chosen threshold. If a threshold is chosen conservatively as in PATCH-SIM, then many incorrect patches are not filtered out. Meanwhile, if a threshold is determined more aggressively as in ODS, then many correct patches are also filtered out (see Section 6.2.1).

In this work, we use an evidence-based approach that does not suffer from the threshold problem; we reject a patch only when concrete evidence for rejection is found via fuzzing, thus guaranteeing high precision. Note that existing evidence-based approaches [11, 59] rely only on program crashes as concrete evidence. However, using only implicit oracles is not enough, and the current evidence-based approaches suffer from low recall [51].

To provide more help to developers, we generalize the evidence-based approach by looking for *any* kind of output discrepancies between patched and pre-patched versions. To look for a discrepancy, we start by generalizing a given failing test. Figure 2(c) shows the failing test for our example buggy version, and Figure 2(d) shows how we generalize the three constants of the existing test into three parameters d1, d2, and d3. Then, using a QuickCheck framework [7] such as junit-quickcheck [14], we can obtain various random values for these parameters to perform differential fuzzing. One remaining problem is that *not all* output discrepancies evidence the incorrectness of the patch, since certain output changes are expected with the patch. To resolve this problem, the oracle also needs to be generalized. In an attempt to generalize the original assertion, 0.975-1.0e-5 <= x <= 0.975+1.0e-5 (lines 6–7 of Figure 2(c)), the user can consult the API document of the method under test, inverseCumulativeProbability, which specifies the following [10]:

— inf{x in R | P(X <= x) >= p} for 0 < p <= 1,
— inf{x in R | P(X <= x) > 0} for p = 0,

where inf, R, and p represent infimum, a set of real numbers, and the parameter to the inverseCumulativeProbability method, respectively. Unfortunately, this mathematical specification is *not executable* and cannot be used directly as an oracle.

In this article, we propose an alternative way to specify the user's intention when validating patches. In the next section, we describe our semi-automatic patch validation methodology named Poracle.

4 OUR APPROACH: PORACLE

Figure 3 illustrates our approach named Poracle. We consider a scenario where an APR tool generates multiple plausible patches for a given buggy program, and the developer needs to review

[4] https://bitbucket.org/maxpei/jaid/wiki/Home

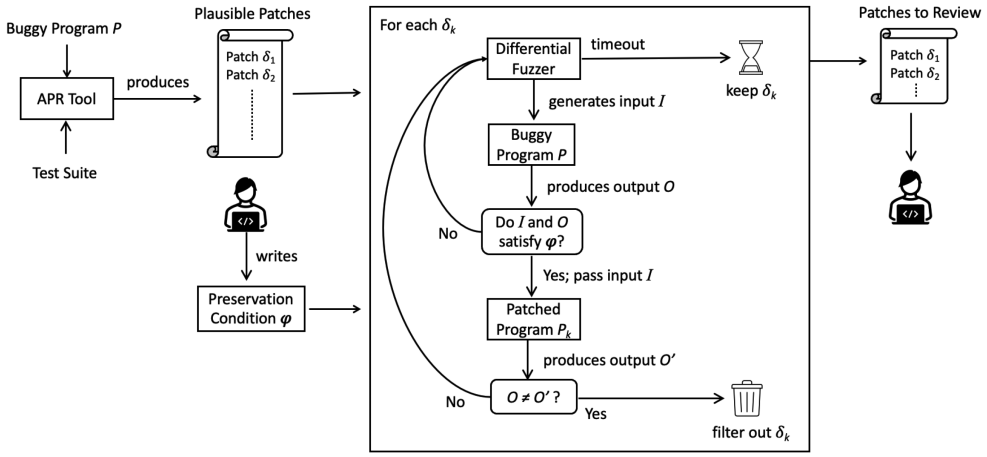


Fig. 3. Our semi-automatic approach filtering out incorrect patches.

those patches to find out the correct one. For each patch δ_k, we take the following steps to decide whether to show it to the developer for review:

(1) We generate an input I using our differential fuzzer described in Section 4.4.
(2) We run the buggy program P with I to produce an output O.
(3) We check whether I and O satisfy the developer-specified preservation condition φ. If so, then we run the patched program P_k with I to produce an output O'. Otherwise, we go back to step 1.
(4) We check whether O and O' are different. If so, then we filter out the patch δ_k. Otherwise, we go back to step 1.
(5) We repeat steps 1–5 until either δ_k is filtered out or a timeout occurs, in which case, we show δ_k to the developer for review.

In this section, we describe our approach in detail. We describe the concept of a preservation condition in Section 4.1 and show in Section 4.2 how preservation conditions can be expressed in existing tests. We provide examples of preservation conditions in Section 4.3. Section 4.4 describes our differential fuzzer. Last, in Section 5, we discuss various factors that can affect the effectiveness of our approach.

4.1 Generalizing a Failing Test with a Preservation Condition

Given a failing test an APR tool used to generate patches (e.g., Figure 2(c)), our specification methodology takes the following two steps:

Step (1) 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), as shown in Figure 2(d). An obtained parameterized test can be perceived as a **parameterized unit test (PUT)** [50] or a **property-based test (PBT)** [7, 14].

Step (2) Generalizing an oracle. Given a parameterized test $T(\vec{x})$ where $\vec{x}$ represents parameters, we need an oracle that can tell whether $T(\vec{x})$ returns a correct output when test T is executed over a patched version with an arbitrary input. A conventional method is to write a formal specification

```
1  public void testGcd(int i, int j) {
2    boolean preservationCondition =
3      !( (i==Integer.MIN_VALUE && j==0) || (i==0 && j==Integer.MIN_VALUE) );
4    preserveIf(preservationCondition, () -> new Long[] { MathUtils.gcd(i, j) });
5  }
```

Fig. 4. An example of a preservation condition.

as shown in Reference [37]. In such an approach, an oracle function $\psi(\vec{x})$ satisfying the following is specified by the user:

$$\forall \vec{v} : T(\vec{v}) = \psi(\vec{v}), \tag{2}$$

where $T(\vec{v})$ represents the output of test T when inputs $\vec{v}$ is assigned to parameters $\vec{x}$. However, as described with the motivating example, writing such ψ is not trivial even if a developer has a perfect understanding of the program under testing. Alternatively, the user may describe a metamorphic relation (such as $decrypt(encrypt(x)) = x$). However, it is usually not easy to find a metamorphic relation effective at bug finding [6, 31].

Generalizing an oracle with 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 should satisfy the following:

$$\forall \vec{v} : \varphi(\vec{v}) \Rightarrow T_P(\vec{v}) = T_{P'}(\vec{v}), \tag{3}$$

where $T_P(\vec{v})$ and $T_{P'}(\vec{v})$ represent the output of test T in P and P', respectively, when inputs $\vec{v}$ is assigned to parameters $\vec{x}$. Here, the intention is that *for any input $\vec{v}$ satisfying φ in the pre-patched version P, the output should be preserved in the patched version P'*. Conversely, if output difference is observed for input $\vec{v}$ satisfying φ in P, then $\vec{v}$ is evidence for the incorrectness of P'. We call Equation (3) a *preservation invariant*.

4.2 Expressing a Preservation Condition in a Test

To see how we express a preservation condition in a test, consider an example of Figure 4 where a bug occurs because method gcd fails to handle a corner case input correctly; when gcd takes as input a pair of Integer.MIN_VALUE and 0, the buggy version returns a wrong value. In this case, developers would want to test whether program behavior is preserved in the non-corner cases (i.e., input is *not* a pair of Integer.MIN_VALUE and 0), and this preservation condition is expressed in line 4 of our example.

As described in Equation (3), we run our example test with both pre-patched and patched versions when the preservation condition holds. To compare the output values of the two versions, we use our custom method preserveIf, as shown in line 4. If the first parameter holds true, then the second parameter is evaluated in both versions, and their values are compared with each other. If different values are observed, then the patch under consideration is classified as incorrect. We use a lambda expression (i.e., () -> . . .) in the second parameter of the preserveIf method for a technical reason.[5]

[5]If an output value is obtained by calling a method (e.g., the method under test returns a reference type value v and to obtain an output, v.toString() is used), then an exception can be raised while calling that method. To ignore such a case, we pass a lambda expression to the preserveIf method.

Table 2. Distribution of Preservation Condition Patterns

Project	UE	CC	EGA	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

We formally define the semantics of preserveIf as follows, where P and P' represent the pre-patched and patched versions, respectively. Note that we run the pre-patched version before running the patched version.

$$P' \vdash \text{preserveIf}(\varphi, \lambda) = \begin{cases} P' \vdash \text{Assert}(\text{eval}(\lambda) == v) & \text{if } P \vdash \text{eval}(\varphi) == true, P \vdash \text{eval}(\lambda) == v \\ P' \vdash \text{nop} & \text{otherwise,} \end{cases} \quad (4)$$

where notation $P' \vdash \text{preserveIf}(\varphi, \lambda)$ denotes that preserveIf(φ, λ) is executed under the patched version P'. Similarly, notation $P \vdash \text{eval}(\varphi) = true$ denotes that preservation condition φ is evaluated to $true$ under the original version P. Notation $P \vdash \text{eval}(\lambda) == v$ is interpreted similarly.

If random input assigned to the parameters (i.e., i and j in the running example) satisfies the preservation condition in the original (pre-patched) version, then we also run the same test with the patched version to compare the output between the versions. Otherwise, we skip running the patched version with the current random input (since $P' \vdash$ nop, there is no need to run the patched version).

4.3 Examples of Preservation Conditions

In this section, we show four examples of preservation conditions. These examples exhibit four different patterns of preservation conditions that cover all buggy versions in our dataset. As detailed in Section 6.1, we write preservation conditions to cover all 77 real-world bugs in our dataset. We obtain these 77 buggy versions from the previous work on patch classification [56]. Table 2 shows the distribution of the four patterns over the 77 buggy versions.

The distribution of the four patterns is shown in Table 2 and we present the examples in the order of the coverage of each pattern.

4.3.1 Unexpected Exception (UE). The test shown in our motivating example—Figure 2(c)—fails, because an exception is thrown when it is not expected. In this case, the developer would want to preserve the program behavior as long as the original version does not throw an exception. Figure 5 shows how we generalize the test. The preserveIf method at line 6 is called only when an exception is not thrown before reaching that line. Thus, the preservation condition is simply true in this case.

What if an exception occurs only in the patched version? In this example, it would be reasonable to classify the patch as incorrect in that case. To do that, we use another custom method failToPreserve at line 9 whose semantics is defined as follows, where *msg* refers to a predefined unique message:

$$P \vdash \text{failToPreserve}() = \begin{cases} P \vdash \text{preserveIf}(\text{true}, msg) & \text{if } P \text{ is a patched version} \\ P \vdash \text{nop} & \text{if } P \text{ is a pre-patched version.} \end{cases}$$

```
1  public void testSmallDegreesOfFreedom(double d1, double d2, double d3) {
2    try {
3      FDistributionImpl fd = new FDistributionImpl(d1, d2);
4      double p = fd.cumulativeProbability(d3);
5      double x = fd.inverseCumulativeProbability(p);
6      preserveIf(/* preservation condition */true, /* outputs to compare */ () -> new Double[] {x});
7    } catch (Exception e) {
8      // If an exception occurs only in the patched version, the patch is classified as incorrect.
9      failToPreserve();
10   }
11 }
```

Fig. 5. Generalized test for Math95 (UE).

```
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    */
8    // Generalized body:
9    try {
10     boolean complement = !( (i==Integer.MIN_VALUE && j==0) || (i==0 && j==Integer.MIN_VALUE) );
11     final long actual = MathUtils.gcd(i, j);
12     preserveIf(complement, () -> new Long[] { actual });
13   } catch (ArithmeticException e) {
14     preserveIf(!complement, () -> new String[] { e.toString() });
15   } catch (Exception e) {
16     failToPreserve();
17   }
18 }
```

Fig. 6. Generalized test for Math99 (CC).

If an exception occurs only at the patched version, then the output of the patched version is *msg*, whereas the output of the pre-patched version is the value of x obtained at line 6. Thus, the discrepancy in the output is detected as desired.

Comparison with the existing approaches. The existing approaches such as Opad [59] and Fix2Fit [11] detect crashing patches that can be viewed as the UE pattern. However, these existing approaches cannot detect *unexpected non-crashing differences* between the versions. In comparison, the test shown in Figure 5 can be used to detect such unexpected non-crashing differences. For example, if method inverseCumulativeProbability behaves differently between pre-patched and patched versions, the outputs of the two versions are compared to each other by calling the preserveIf method.

By using the generalized test shown in Figure 5, *we succeed to reject all 15 incorrect patches available in our benchmark while accepting the correct patch (Figure 2(b)).*

4.3.2 ***Complementary Cases (CC).*** A software fault often occurs when corner-case behavior is not yet implemented. Consider Figure 6 whose simplified version was shown earlier in Figure 4. In this example, the bug occurs because method gcd fails to handle a corner case input correctly;

```
1  public void testSSENonNegative(double d1, double d2, double d3, double d4, double d5, double d6) {
2    try {
3      double[] y = { d1, d2, d3 };
4      double[] x = { d4, d5, d6 };
5      SimpleRegression reg = new SimpleRegression();
6      for (int i = 0; i < x.length; i++) {
7        reg.addData(x[i], y[i]);
8      }
9      double ret = reg.getSumSquaredErrors();
10     // Original: assertTrue(ret >= 0.0);
11     preserveIf(ret >= 0.0, () -> new Double[] { ret });
12   } catch (Exception e) {
13     failToPreserve();
14   }
15 }
```

Fig. 7. Generalized test for Math105 (EGA).

when gcd takes as input a pair of Integer.MIN_VALUE and 0, an ArithmeticException is expected to be thrown (see line 5), 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 6 can detect an incorrect patch that returns an incorrect output (line 12). It can also detect incorrect patches that either fail to throw an expected ArithmeticException (line 14) or throw an unexpected exception (line 16).

4.3.3 ***Existing General Assertion (EGA).*** A certain assertion existing in the test can be repurposed as a preservation condition. For example, consider Figure 7 where the original test uses an assertion condition, ret >= 0.0 (line 10). Note that this assertion should be satisfied for all input values (i.e., d1, d2, . . ., d6) as long as no exceptions are thrown. Many tests, if not all, contain such general assertions that should be satisfied for all inputs. In such cases, the assertion can be reused as a preservation condition.

Note that reusing the existing condition as a preservation condition *does not mean that oracle power stays the same*. Note that incorrect patches assigning wrong positive values to variable ret still satisfy ret >= 0.0. Those incorrect patches cannot be rejected if ret >= 0.0 is solely used as an oracle. Only after using ret >= 0.0 as a preservation condition, the oracle power is elevated; the aforementioned incorrect patches can be detected, since outputs will differ between the two versions.

4.3.4 ***Reference Implementation (RI).*** It is known that developers often write redundant implementations [2, 3, 17]. Consider Figure 8 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 the patch if the method under test returns the same output as the reference implementation in the pre-patched version.

4.4 Differential Fuzzing

In this work, we advocate a semi-automated approach for patch validation. Once a preservation condition is written, the next step—finding an input that makes the patched version violate the specified preservation invariant—is automatically performed via differential fuzzing.

We design a fuzzer that takes as input a pair of pre-patched and patched versions and a generalized test. Specifically, we customize JQF [40], a coverage-guided fuzzer for Java, so our two

```
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 {
7      actOut = "" + NumberUtils.createNumber(s).longValue();
8    } catch (Exception e) {
9      actOut = "Exception";
10   }
11   // Use Long.valueOf as a reference
12   String refOut = "";
13   try {
14     refOut = "" + Long.valueOf(s);
15   } catch (Exception e) {
16     refOut = "Exception";
17   }
18   preserveIf(actOut.equals(refOut), () -> new String[] { actOut });
19 }
```

Fig. 8. Generalized test for Lang58 (RI).

specification APIs (preserveIf and failToPreserve) are supported. In addition, we add features for differential fuzzing to JQF, as these features are not supported in the original JQF. Our differential fuzzing tool can be obtained at https://github.com/PLaSE-UNIST/poracle-tool.

Algorithm 1 shows our fuzzing algorithm. The overall structure follows the standard coverage-guided fuzzing. The algorithm maintains a set of interesting input S. Set S is initialized with an empty set (line 2), and random inputs are used until S becomes non-empty (line 10). If new interesting input is found, then the current input is added to S (line 32). The meanings of the Energy (line 14) and ShouldSave (line 31) functions are explained later in this section. In JQF, coverage is a set of all coverage points (e.g., program branches) covered by the input. In our fuzzer, we extend this concept and concatenate the coverage obtained from both pre-patched and patched versions.

While the overall structure of our fuzzer follows the standard coverage-guided fuzzing, we also customize our fuzzer as described below.

4.4.1 Considering the Specified Preservation Condition. Unlike the conventional differential fuzzers, our fuzzer does not have to always execute both pre-patched and patched versions. Instead, our fuzzer runs a patched version only when a given preservation condition φ is satisfied in its pre-patched version (lines 18–30). This is because if φ is not satisfied in the pre-patched version, the given preservation invariant cannot be violated. Thus, when φ is not satisfied in the pre-patched version, we directly generate the next input and run the pre-patched version.

When generating a random input, our fuzzer chooses a value in the range of $[c-\delta, c+\delta]$ where c is a fixed constant and δ is chosen *adaptively*.[6] In general, the fuzzing space grows as a larger range is used, and as a result, fuzzing efficiency decreases. To balance the fuzzing space and efficiency, our fuzzer gradually widens the range until an input satisfying a given preservation condition is found. More specifically, we widen the range if the given preservation condition is not satisfied for

[6]The value of c can be obtained from the constant value used in the original failing test. For example, the value of parameter d3 of Figure 2(d) is in the range $[0.975 - \delta, 0.975 + \delta]$.

ALGORITHM 1: The PORACLE fuzzing algorithm

Input: a buggy version p and its patched version p'
Input: a parameterized test with a preservation condition φ
Output: witness value $\vec{v}$ that makes the patched version violate the specified preservation invariant

1: $\vec{v} \leftarrow \bot$
2: $S \leftarrow \emptyset$ ▷ a set of interesting input
3: $totalCov \leftarrow \emptyset$ ▷ total coverage
4: $rangeFixed \leftarrow false; noProgress \leftarrow 0$
5: **repeat**
6: **if** $S = \emptyset$ **then**
7: **if** $noProgress >= T$ **then**
8: WIDENRANGE( ); $noProgress \leftarrow 0$
9: **end if**
10: $s \leftarrow r$ ▷ r: a random value within in the range
11: **else**
12: $s \leftarrow$ CHOOSENEXT(S)
13: **end if**
14: **for** i from 1 to ENERGY(s) **do**
15: $s^{\dagger} \leftarrow$ MUTATE(s, S)
16: // Run the original version p
▷ o: outputs, cov: coverage, σ: program states
17: $o, cov, \sigma \leftarrow$ RUN($p, s^{\dagger}$)
18: **if** $\varphi(s^{\dagger}, o)$ **then**
19: // Preservation condition φ is satisfied
20: $noProgress \leftarrow 0$
21: // Run the patched version p'
▷ σ': dumped program state
22: $o', cov', \sigma' \leftarrow$ RUN($p', s^{\dagger}$)
23: **if** $o \neq o'$ **then**
24: $\vec{v} \leftarrow s^{\dagger}$
25: **return** $\vec{v}$ ▷ return a found witness value
26: **end if**
27: **else**
28: // Preservation condition φ is not satisfied
29: $noProgress$++; $cov' \leftarrow \bot$
30: **end if**
31: **if** SHOULDSAVE($cov, cov', s^{\dagger}$) **then**
32: $S \leftarrow S \cup \{s^{\dagger}\}$
33: **end if**
34: **end for**
35: **until** timeout reached

T consecutive times (line 8).[7] In cases where a fixed range should be used for a certain parameter (e.g., a parameter representing the week of a date), we allow developers to express that intention, and adaptive widening is disabled.[8]

4.4.2 Considering the Program State Changes between the Two Versions. To trigger an output difference between the pre-patched and patched versions, it is necessary to propagate state changes

[7]We use 10 in our experiments.
[8]This feature is not shown in Algorithm 1 for the simplicity of presentation.

made in the patched location—we assume that a single program location is patched as done in most current APR tools—toward the end of the execution path. To achieve this, our fuzzer exploits not only code coverage but also the program state changes observed between the two versions. More concretely, we add an input $s^{\dagger}$ to a set of interesting input S (i.e., ShouldSave returns true) when one of the following two conditions is satisfied:

(C1) $s^{\dagger}$ covers a new branch in either the pre-patched or patched version.
(C2) $s^{\dagger}$ propagates the state changes made in the patched location *further* toward the end of the execution path.

To check C2, we do the following: First, when the patched location is reached while executing the patched version, we extract the stack trace, $[m_1, m_2, \ldots, m_k]$, where m_1 refers to the patched method, m_2 is the caller of m_1, and m_k is the top-level method. Then, at each exit point of m_i, we extract a program state σ_i and σ_i' from the buggy version and its patched version, respectively. We define a program state as a set of mappings from a variable to its value. To confine the size of the program state, we keep track of only the variables v satisfying the following properties. Below, we denote the receiver object of m_i as r. We do not consider static methods.

— v is reachable from r or refers to the ghost variable holding the return value of m_i.
— The type of v is either primitive or String.

We then compute the distance between σ_i and σ_i' as the summation of $|v - v'|$ for each variable v in σ_i and its corresponding variable v' in σ_i'. When v has the String type, we compute the Levenshtein distance between v and v'.

Finally, given the stack trace $[m_1, m_2, \ldots, m_k]$, we compute a distance list $[d_1, d_2, \ldots, d_k]$ where d_i denotes the state distance between σ_i and σ_i'. Note that if d_i is zero, then d_j for $j > i$ is also zero. We consider that C2 is satisfied when one of the following conditions holds:

(1) d_j is positive for the first time—i.e., the state changes are propagated further than before.
(2) d_j is larger than its previous maximum value and d_{j+1} is zero in all cases so far including the current case.

Among the saved inputs, our fuzzer prioritizes those that propagate state changes further. To achieve this, we sort the saved inputs as follows:

— For input s, let $max\_idx(s)$ denote the maximum index j such that d_j is positive. We place an input s_1 after s_2 when $max\_idx(s_1) > max\_idx(s_2)$.
— If $max\_idx(s_1) = max\_idx(s_2)$, then we break the tie as follows: Let $max\_ids\_val(s)$ denote the value of d_j where max $\_idx(s) = j$. We place an input s_1 after s_2 when $max\_idx\_val(s_1) > max\_idx\_val(s_2)$.

Once sorting is done, we assign energy to input s (i.e., the number of iterations s is mutated). In Algorithm 1, the Energy function returns the energy assigned to s. To mutate more promising input—i.e., an input towards the end of the sorted input list—more frequently, we use the following formula: $MaxEnergy \times \frac{i+1}{|S|}$ where $MaxEnergy$ refers to the maximum value Energy can return and i is the index of s in the sorted input list S.

5 DISCUSSION

Our method is specifically designed for developers who desire greater control over the patch classification process. We assume that the users of our approach have adequate domain-specific knowledge about the target program and the patch. Please note that our method is not intended to replace existing automatic PC techniques. Rather, it fills a gap the existing PC techniques cannot cover. Even if developers wish to incorporate their domain-specific knowledge, the existing PC techniques do

not allow for this. Our approach provides developers with a language to express their domain-specific knowledge (i.e., preservation condition) and a tool (i.e., a differential fuzzer) to filter out incorrect patches based on this knowledge.

We discuss the benefits and limitations of our semi-automatic approach in Sections 5.1 and 5.2, respectively. Subsequently, we present in Section 5.3 a guideline to write a preservation condition to minimize the limitation of our approach. In Section 5.4, we discuss the possibility of further automating our approach. Finally, Section 5.5 compares our approach with conventional test assertions.

5.1 Benefits of Our Approach

APR systems are expected to be integrated into a **continuous integration (CI)** system. Imagine that one of the regression tests fails, and an APR tool generates a set of patches. In case it is clear how to fix the detected bug, the developer would easily fix the bug without having to take a look at the generated patches. APR and our patch validation approach can be handy in the opposite case. If the developer is not sure about how to fix the bug and an APR tool generates multiple patches, then the developer would consider using our approach to obtain the following benefits: First, as shown in this work, incorrect patches an APR tool generates can be filtered out. This can reduce the cognitive load of the developer, since multiple incorrect patches can be filtered out by generalizing a failing test with a preservation condition only once.

As the second benefit, the developer can have higher confidence in the obtained patch surviving hundreds of tests instantiated from the prepared generalized test. In the current APR community, it is generally assumed that the developer can easily determine the correctness of a patch an APR system generates. However, recall that the fault localization research community initially made a similar assumption—simply suggesting a faulty line candidate would be enough for the developer to detect, understand, and fix the bug—but, since the seminal paper of Parnin and Orso [42], this assumption was abandoned in the community. We argue that the APR community should also consider the possibility that simply suggesting a patch may not be sufficient for the developer to decide the correctness of a patch. Indeed, earlier studies show that developers often make mistakes when writing a patch [21, 34]. Our approach helps the developer identify incorrect patches.

Last, once a generalized test is written, it can be reused to test future code changes as long as the preservation condition does not change. When a test fails in a CI system and as a follow-up, an APR system generates multiple patches, only the patches that survive all available generalized tests will be shown to the user.

While it takes some time for a user to extend the existing failing test into a generalized test, this task can be done while an APR system is running in the CI system, given that it takes a long time for an APR tool to generate a patch. The time spent preparing a generalized test can be compensated in multiple ways as described.

5.2 Limitations of Our Approach

5.2.1 Impact of Imperfect Preservation Condition. As with any approach that involves human input, the effectiveness of our approach depends on the quality of the input provided. There are the following four different cases to consider:

(1) A user-written preservation condition φ is equivalent to the ground-truth preservation condition ψ (i.e., $\varphi = \psi$).
(2) φ is an under-approximation of ψ (i.e., φ is stronger than ψ).
(3) φ is an over-approximation of ψ (i.e., φ is weaker than ψ).
(4) φ is wrong (i.e., neither φ nor ψ completely includes the other).

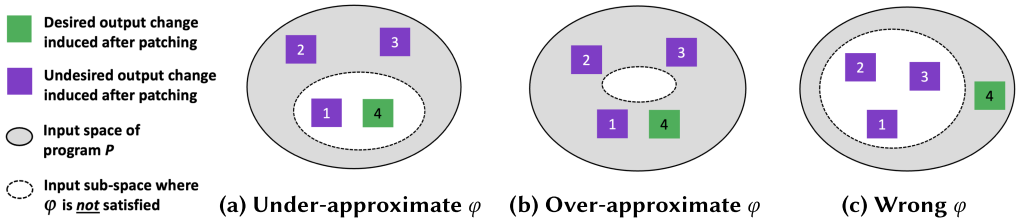


Fig. 9. Imperfect preservation conditions.

Figure 9 illustrates the latter three cases where φ is not equivalent to ψ. In the figure, each shape represents the following:

- **The outer solid ellipse.** It represents the input space S of the original buggy version P.
- **The inner dotted ellipse.** It represents the sub-space of S where preservation condition φ is *not* satisfied. Thus, φ is satisfied in the grey donut-shaped space.
- **Three purple squares.** They represent the three undesired output changes induced between the original buggy version P and the patched version P'. We use the notation $\delta_i^{\text{✗}}$ (where $1 \leq i \leq 3$) to refer to these three purple squares.
- **The green square.** It represents the desired output change induced between P and P'. We refer to this green square by the notation $\delta_4^{\checkmark}$.

We classify the patched version P' as incorrect if a square of any color is detected in the grey area. This leads to the following consequences:

- **Under-approximated φ.** This amounts to Figure 9(a). The patch is correctly classified as incorrect if $\delta_2^{\text{✗}}$ or $\delta_3^{\text{✗}}$ is detected. However, if $\delta_1^{\text{✗}}$ is detected, then our method does not classify the patch as incorrect.
- **Over-approximated φ.** This amounts to Figure 9(b). The patch is correctly classified as incorrect if $\delta_1^{\text{✗}}$, $\delta_2^{\text{✗}}$, or $\delta_3^{\text{✗}}$ is detected. The detection of $\delta_4^{\checkmark}$ also results in the correct classification of the patch as incorrect. However, this classification is only coincidentally correct.
- **Wrong φ.** This amounts to Figure 9(c). The patch is incorrectly classified as incorrect if $\delta_4^{\checkmark}$ is detected.

In summary, using an under-approximate preservation condition can lead to false negatives (i.e., an incorrect patch is not filtered out), while using an over-approximate preservation condition can lead to false positives (i.e., a correct patch is filtered out).

5.2.2 Non-regression Error. Our approach can detect a regression error caused by a patch. If the specified preservation condition is met, then the output difference between the original buggy version and the patched version is considered a regression error, leading to the classification of the patch as incorrect. However, regression is not the only reason for a patch to be incorrect. A patch is also incorrect if it does not change the output of the original version correctly. While our approach uses the original version as the oracle when a given preservation condition is met, it does not check the behavioral correctness when the preservation condition is not met, which may result in failing to detect incorrect patches.

5.3 A Guideline to Write a Preservation Condition

The efficacy of our approach depends on the quality of the preservation condition. However, this does not mean that developers need to write a complex preservation condition. On the contrary, when we applied our method to automatically generated patches for real-world bugs (Section 6),

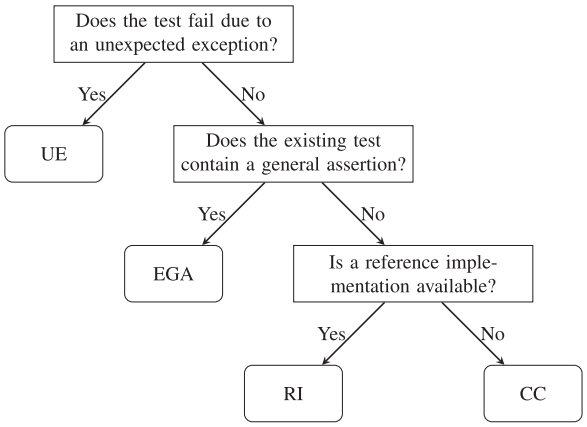


Fig. 10. Decision tree about preservation condition patterns.

we found that a simple preservation condition is often sufficient (see Figure 16). In this section, we discuss a possible guideline for writing a preservation condition.

A preservation condition can be viewed as an answer to the following question: *What are the conditions under which I can confidently trust the correctness of the output produced by the original version?* For example, if the bug at hand is about an exception unexpectedly thrown, the developer would assume that the output of the original version is correct when that exception is not thrown.

In Section 4.3, we introduced four preservation condition patterns: UE, CC, EGA, and RI. All these patterns provide a framework for answering the aforementioned question of when one can trust the correctness of the output produced by the original version. If a developer wants to directly specify the condition under which the bug at hand occurs, then they can use the CC pattern. Note that a preservation condition is the negation of the bug-triggering condition that has been identified. However, while the CC pattern is most general, it often leads to a more complex preservation condition compared to the other patterns (see the "Pattern Complexity" column of Table 10). Therefore, we recommend that developers consider using the other patterns first if possible, as they can be viewed as special cases of the CC pattern.

The UE pattern can be used when the bug is manifested by an unexpected exception, while the EGA pattern can be used when the existing test already contains a general assertion. Similarly, the RI pattern can be used when a reference implementation is available. To provide developers with a concrete guideline on how to write a preservation condition, we offer a decision tree shown in Figure 10.

5.4 Discussion about Automation

Given the necessity of domain-specific knowledge, it would not be easy to fully automate the test generalization process. However, the process can be partly automated in a similar way to *refactoring* incorporated into IDEs such as IntelliJ [15] and Visual Studio Code [35]. For example, when the user observes an unexpected exception from a test, the automatic transformation of an existing test (e.g., Figure 2(c)) into a generalized version (e.g., Figure 5) can be performed in a straightforward manner. For the remaining less common patterns, more user input is necessary, similar to some refactoring patterns. Once the user chooses a generalization pattern to use and a necessary preservation condition, the rest of the transformation can be performed automatically, after which the obtained transformed test can be reviewed/edited by the user. Leaving IDE integration as future work, we first investigate the efficacy of generalized tests in filtering out incorrect patches.

5.5 Preservation Conditions vs. Conventional Test Assertions

In Section 4.1, we mentioned that writing an oracle function $\psi(\vec{x})$ is difficult even if a developer has a perfect understanding of the program under testing. Recall that $\psi(\vec{x})$ satisfies the following:

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

where $T(\vec{v})$ represents the output of test T when inputs $\vec{v}$ is assigned to parameters $\vec{x}$ of the generalized test. Conventionally, ψ is written as a test assertion.

We also showed a concrete example (i.e., the motivating example shown in Section 3) for which it is not clear how to write a conventional assertion, whereas the preservation condition for the same example is very simple (i.e., `true`), as shown in Figure 5.

As another example, compare the two generalized tests shown in Figures 11. Notice that Figure 11(a) uses conventional assertions, whereas Figure 11(b) uses a preservation condition. The generalized test shown in Figure 11(a) expresses the following requirements of the `gcd` method [9]:

(1) Lines 4–5: The invocation `gcd(0, 0)` is the only one that returns 0.
(2) Lines 11–16: Computes the greatest common divisor of the absolute value of two numbers.
(3) Lines 6–9: The result of `gcd(x, x)`, `gcd(0, x)`, and `gcd(x, 0)` is the absolute value of `x`, except for the special cases above.
(4) Lines 19–20: The invocations `gcd(Integer.MIN_VALUE, Integer.MIN_VALUE)`, `gcd(Integer.MIN_VALUE, 0)`, and `gcd(0, Integer.MIN_VALUE)` throw an `ArithmeticException`.

As shown with this example, when writing a conventional test, the test writer often needs to understand how to partition the input space and write an oracle for each input partition. In comparison, writing a preservation condition is simpler for the following two reasons:

— First, as illustrated in Figure 12(b), the writer of a preservation condition needs to split the input space into only two partitions, one where the behavior should be preserved after patch and the other one where the behavior is allowed to be changed.
— There is no need to write an assertion to check behavioral preservation, because the output of the original version serves as the oracle.

Bug fixing typically involves making limited changes to the program behavior, and this makes it suitable for describing when the program behavior should be preserved.

6 ASSESSMENT

We assess our specification-based patch-validation approach, Poracle, with the following research questions:

— **RQ1 (Classification Performance):** How does our approach perform as compared to the state-of-the-art approaches?
— **RQ2 (Other Attributes):** Apart from classification, we also evaluate our approach in terms of other important attributes listed in the following:
  — **RQ2-1 (Consistency):** Poracle uses a fuzzing technique, which may not produce the same result across multiple runs. How does the randomness of fuzzing affect the consistency of the results across multiple runs?
  — **RQ2-2 (Time efficiency):** If a patch classification takes too long, then it would impede APR adoption by practitioners. How fast does our fuzzer detect incorrect patches?
  — **RQ2-3 (Ablation Study):** Our patch-classification approach consists of two parts: (1) generalizing a failing test with preservation conditions and (2) conducting differential fuzzing. To evaluate the impact of preservation conditions on classification performance, we

```
1  public void testGcd(int i, int j) {
2    try {
3      long actual = MathUtils.gcd(i, j);
4      if (i == 0 && j == 0)
5        assertEquals(0, actual);
6      else if (i == j || j == 0)
7        assertEquals(Math.abs(i), actual);
8      else if (i == 0)
9        assertEquals(Math.abs(j), actual);
10     else {
11       // check if actual is the true gcd.
12       if (i % actual != 0 || j % actual != 0)
13         fail();
14       for (int k = actual + 1; k < Math.max(Math.abs(i), Math.abs(j)); k++)
15         if (i % k == 0 && j % k == 0)
16           fail();
17     }
18   } catch (ArithmeticException e) {
19     assertTrue((i == MIN_VALUE && j == Integer.MIN_VALUE) || (i == Integer.MIN_VALUE && j == 0)
20                || (i == 0 && j == Integer.MIN_VALUE));
21   }
22 }
```

(a) Generalized test for Math99 using the conventional test assertions

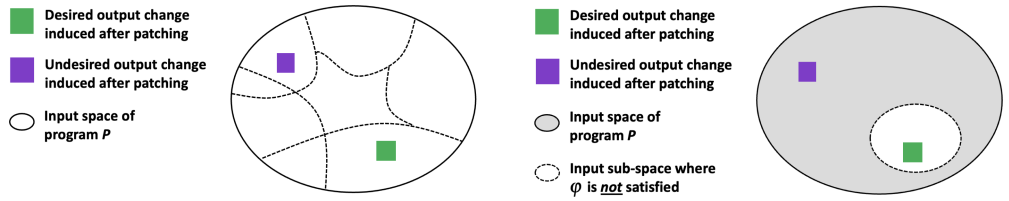
```
1  public void testGcd(int i, int j) {
2    // Generalized body:
3    try {
4      boolean complement = !( (i==Integer.MIN_VALUE && j==0) || (i==0 && j==Integer.MIN_VALUE) );
5      final long actual = MathUtils.gcd(i, j);
6      preserveIf(complement, () -> new Long[] { actual });
7    } catch (ArithmeticException e) {
8      preserveIf(!complement, () -> new String[] { e.toString() });
9    } catch (Exception e) {
10     failToPreserve();
11   }
12 }
```

(b) Generalized test for Math99 using a preservation condition (copied from Figure 6)

Fig. 11. Two kinds of generalized tests for Math99.

perform an ablation study by disabling preservation conditions. We compare the obtained results with those obtained using the full approach.

  - **RQ2-4 (Application scope):** APR research is gradually moving towards more complex patches [53]. Is Poracle effective for more complex patches?
- **RQ3 (Cost reduction):** Our work is motivated by the high cost of manually validating a large number of plausible patches. When our approach is applied to an APR system that returns a ranked list of plausible patches, how much cost reduction can be achieved by filtering out incorrect patches?
- **RQ4 (Usability):** Poracle uses a semi-automated approach, and thus, we study how the user adopts our approach.



(a) When conventional test assertions are used. Input space is divided into multiple partitions, some of which may overlap, with each partition having its own associated oracle in a form of an assertion.

(b) When a preservation condition is used. Input space is divided into two partitions and when the preservation condition is satisfied (i.e., the grey region), the output of the original version is used as an oracle.

Fig. 12. Comparison of output spaces when conventional test assertions and a preservation condition are used.

6.1 Experimental Settings

In this subsection, we describe the experimental settings we used for RQ1 (Classification Performance) and RQ2 (Other Attributes). For the remaining two RQs, we conduct different experiments, as will be described later.

Datasets. Our dataset contains in total 458 patches collected in the following way: We start with the 139 patches used in PATCH-SIM [56], a de facto standard dataset [49, 51, 60] for PC techniques. For a fair comparison, we run the first experiment with the PATCH-SIM dataset prepared by the authors of PATCH-SIM. The PATCH-SIM dataset consists of the 139 patches generated from 77 buggy versions in the Defects4J benchmark [19] and their correctness labels. For each of those 77 buggy versions, we generalize its failing test by adding a preservation condition. In case more than one failing test exists, we generalize only one failing test, simulating a bug-fixing scenario where a developer typically has only one failing test initially. Note that most bugs (61 out of 77) in our dataset have only one failing test.

We also run the second experiment with an extended dataset extracted from the large-scale work of Liu et al. [27] where the authors collected patches generated from 16 APR systems and labeled the correctness of each patch after manual investigation. A patch is labeled as correct when it is identical to or semantically similar to the developer-written patch. For the criteria used to determine semantic similarity, please refer to Reference [27]. Labeling the correctness of a patch is a laborious and error-prone task, and we utilize the aforementioned existing labeled dataset. Using the labels prepared by other researchers also helps us mitigate the internal threat to validity caused by wrong or prejudiced labels. We collect all patches in Reference [27] generated from the buggy versions covered by our 77 generalized tests. This step adds 350 patches to our pool of patches. In the end, we obtain 458 patches after removing 31 duplicate patches that exist in both datasets. Table 3 summarizes our dataset where the notation x+y denotes x patches labeled as incorrect and y patches labeled as correct. Note that the extended dataset is the superset of the PATCH-SIM dataset. In the last column of Table 3, the number of developer patches is identical to the number of buggy versions used in our dataset, as our dataset contains one developer patch for each buggy version.

In summary, we use two datasets in our experiments, the PATCH-SIM dataset and the extended dataset, both of which share the same 77 buggy versions.

Patch Classification Techniques. To assess the performance of Poracle, we compare the results with the following four state-of-the-art patch classification techniques: PATCH-SIM [56],

Table 3. Dataset Used in Our Experiment

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

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

OPAD [59], and two machine-learning-based approaches [49, 60], which includes all existing PC techniques targeted for Java programs mentioned in Section 2. Note that anti-patterns [47] and FIX2FIT [11] are designed for C programs, and we exclude them from our experiment.

As mentioned in Section 2, we do not consider PE techniques [55, 58, 61, 65] such as DIFFTGEN [55] in our evaluation. PE techniques assume the existence of ground-truth patches and use them to check whether the patch under consideration is semantically equivalent to its ground-truth patch. The target users of the PE techniques are APR researchers who want to evaluate the quality of the generated patches. In comparison, the target users of the PC techniques like ours are developers in the field; the PC techniques should work without ground-truth patches. Note that while our benchmark contains ground-truth patches, our classification technique does not use them.

Among the four PC techniques, OPAD uses an evidence-based approach, whereas the remaining three tools use score-based approaches. OPAD classifies a patch as incorrect if a crash is detected when a patched version is run under a fuzzer. PATCH-SIM computes the distance between the executions of test t before and after applying the patch p (denoted with $dist_p(t)$) and classifies the patch as incorrect in the following two cases: (1) when the maximum distance observed in the passing tests is larger than the predefined threshold (i.e., after patch, the execution of the passing test substantially deviates from the original execution path) and (2) when the maximum distance observed in the passing tests is larger than the average distance observed in the failing tests (i.e., after patch, the execution path deviates further from the passing tests than from the failing tests). Otherwise, the patch is classified as correct. We will describe the two ML-based approaches shortly.

We run our fuzzer with a 10-min timeout, using our generalized tests as fuzz drivers. We run PATCH-SIM with a 35-min timeout. PATCH-SIM is reported to take up to 30 minutes in its original work [56]. PATCH-SIM internally uses Randoop [39] to prepare test cases to run, and we use a 3-min timeout for Randoop as used in the original experiment of PATCH-SIM. Opad is originally implemented for C programs, and we simulate Opad in Java by classifying a patch incorrect if a patched program throws an exception that is not observed in the original version; the same approach was used in previous work [56]. For a fair comparison with Opad and Poracle, we use the same fuzzer and timeout for both tools (i.e., 10 mins). All our experiments are performed on Intel Xeon Gold CPU and 128 GB memory.

We also compare Poracle with two ML-based approaches [49, 60]. In Reference [49], patch classification is performed by first transforming patch code into an embedding vector using an embedding technique (such as BERT [8]) and then passing an obtained embedding vector to an ML classifier (such as logistic regression), which is trained with the embedding vectors transformed from correct and incorrect patches. In Reference [49], the combination of BERT and logistic regression (denoted with BERT-LR in this article) is shown to perform best, and we use that in our comparison study. While the tool used in Reference [49] is not publicly available, the embedding vectors for the patches in their dataset and the classification script is available, from which we obtain results of BERT-LR for 286 patches existing in common between our dataset and theirs.

Another ML-based tool, ODS [60], also trains a classification model using the XGBoost library [5]. Unlike Reference [49], ODS uses hand-crafted code features. For example, one of the features determines whether a patch uses a local variable or a global variable. Another example of the feature determines whether a patch is to add a missing null check. We compare Poracle with ODS, using 332 patches existing in common between our dataset and theirs.[9]

Evaluation Metrics. We use standard metrics, precision, recall, and F-measure defined as follows:

$$Precision = \frac{\text{(the number of rejected incorrect patches)}}{\text{(the total number of rejected patches)}},$$

$$Recall = \frac{\text{(the number of rejected incorrect patches)}}{\text{(the total number of incorrect patches)}},$$

$$F\text{-}measure = \frac{2}{\frac{1}{Recall} + \frac{1}{Precision}}.$$

6.2 Experimental Results

6.2.1 RQ1: Performance. Table 4 compares the performance of Poracle with PATCH-SIM and Opad. In the table, the "Patches" column shows the number of incorrect/correct patches in our dataset. We use notation x/y where x and y represent the data for the PATCH-SIM dataset and the extended dataset (Table 3), respectively. This distinction is made to show the PATCH-SIM results reported by its authors verbatim [56] for the PATCH-SIM dataset. We use the same notational convention for "Precision," "Recall," and "F-measure" columns to distinguish the results from the two datasets; in notation x/y, x and y represent the result for the PATCH-SIM dataset and the extended dataset, respectively. In each row of the table, the best results are highlighted.

Poracle significantly outperforms PATCH-SIM and Opad across all four projects. The recall of Poracle reaches 60% (197/326), which is about 2 times and 4 times higher than PATCH-SIM (31%)

[9]While the ODS replication package is available (i.e., their benchmark results can be reproduced), the ODS "tool" was not available (i.e., a new benchmark cannot be applied) at the time of conducting the experiments.

Table 4. Comparison between Poracle, PATCH-SIM, and Opad

Project	Patches		Precision			Recall			F-measure		
	Incorrect	Correct	Poracle	PATCH -SIM	OPAD	Poracle	PATCH -SIM	OPAD	Poracle	PATCH -SIM	OPAD
Chart	24 / 24	2 / 2	100% / 100%	100% / 100%	67% / 67%	71% / 71%	58% / 58%	8% / 8%	83% / 83%	73% / 73%	14% / 14%
Lang	11 / 32	4 / 19	100% / 100%	100% / 65%	50% / 50%	82% / 59%	54%/ 34%	9% / 3%	90% / 74%	70% / 45%	15% / 6%
Math	64 / 250	19 / 98	100% / 99%	100% / 96%	81% / 68%	62% / 59%	52% / 26%	27% / 16%	77% / 74%	68% / 41%	41% / 26%
Time	13 / 20	2 / 13	100% / 100%	100% / 100%	100% / 100%	77% / 63%	69% / 50%	54% / 40%	87% / 77%	81% / 67%	70% / 57%
Total	112 / 326	27 / 132	100% / 99%	100% / 92%	82% / 71%	70% / 60%	55% / 31%	24% / 16%	82% / 75%	71% / 46%	37% / 26%

For a fair comparison, we use two different datasets: (1) 139 (=112+27) patches used in the study of PATCH-SIM [56] and (2) the extended dataset shown in 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.

Table 5. Comparison between Poracle and BERT-LR

Project	Patches		Precision		Recall		F-measure	
	Incorrect	Correct	Poracle	BERT-LR	Poracle	BERT-LR	Poracle	BERT-LR
Chart	24 / 24	2 / 2	100% / 100%	100% / 100%	71% / 71%	67% / 67%	83% / 83%	80% / 80%
Lang	11 / 24	4 / 8	100% / 100%	100% / 86%	82% /58%	10% / 25%	90% / 73%	18% / 39%
Math	64 / 167	19 / 42	100% / 99%	100% / 99%	66% / 63%	36% / 44%	80% / 77%	53% / 61%
Time	13 / 16	2 / 3	100% / 100%	100% / 100%	77% / 69%	23% / 25%	87% / 82%	37% / 40%
Total	112 / 231	27 / 55	100% / 99%	100% / 98%	70% / 64%	38% / 43%	82% / 78%	55% / 60%

For a fair comparison, we use two different datasets: (1) 139 (=112+27) patches used in the study of PATCH-SIM [56] (the same as in Table 4) and (2) 286 (=231+55) patches, which are the intersection between the dataset available in Reference [49] and our extended dataset (Table 3).

and Opad (16%), respectively. Also, the precision of Poracle reaches 99%[10] (197/(197 + 2)), which is higher than that of PATCH-SIM (92%) and Opad (71%). Opad, for which we use the *same fuzzer* used in Poracle, shows the *lowest* performance, indicating the importance of using specification. The F-measure is also highest in Poracle across all four subjects.

Table 5 compares the performance of Poracle with BERT-LR [49]. We use the same notation x/y as before. Our result shows that Poracle outperforms BERT-LR as well. The recall of Poracle (64%) is about 1.5 times higher than that of BERT-LR (43%), while the precision is high both in Poracle (99%) and BERT-LR (98%). As a result, the F-measure is higher in Poracle than in BERT-LR.

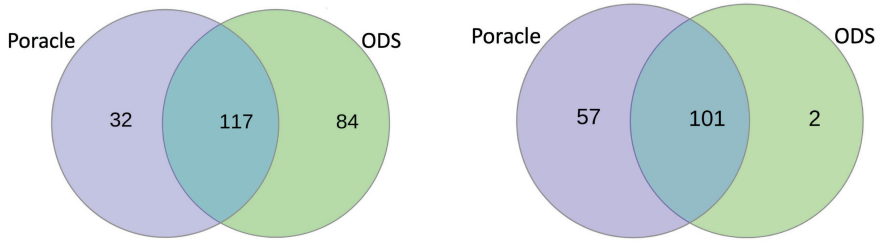
Last, Table 6 compares Poracle and ODS. Poracle shows a higher recall than ODS in the PATCH-SIM dataset, and the opposite result is observed from the other dataset. In terms of precision, Poracle maintains high precision (100% and 99%), whereas ODS shows a lower precision (94% and 88%) in the two datasets. Overall, ODS rejects patches more aggressively, whether they are correct or incorrect. Figure 13 shows such a tendency clearly. As shown in the left diagram, ODS rejects 52 more incorrect patches correctly than Poracle. However, the right diagram[11] shows the strength of Poracle over ODS, accepting 55 more correct patches. Our mixed result, (which is also reflected in the F-measure), poses the classic tradeoff between sensitivity and specificity of classification. However, considering the scarcity of correct patches [29], the cost of erroneously rejecting a correct patch is very high, and our approach is advantageous over ODS in that regard. In fact, rejecting a correct patch should be the last thing a patch validation tool does considering the objective of APR—i.e., finding out a correct patch.

[10]Two correct patches are rejected due to that the imperfect preservation conditions we used. Instead of engineering our preservation conditions, we used the preservation conditions prepared based on our understanding on the bugs.
[11]To compensate for a small number of correct patches, we also include developer patches.

Table 6. Comparison between Poracle and ODS

Project	Patches		Precision		Recall		F-measure	
	Incorrect	Correct	Poracle	ODS	Poracle	ODS	Poracle	ODS
Chart	23 / 23	2 / 2	100% / 100%	100% / 100%	70% / 70%	57% / 57%	82% / 82%	73% / 73%
Lang	10 / 26	3 / 10	100% / 100%	100% / 78%	80% / 58%	90% / 96%	89% / 73%	94% / 86%
Math	60 / 177	19 / 64	100% / 99%	92% / 90%	68% / 61%	55% / 84%	81% / 75%	69% / 87%
Time	13 / 16	2 / 7	100% / 100%	92% / 70%	77% / 69%	85% / 88%	87% / 82%	88% / 78%
Total	106 / 242	26 / 83	100% / 99%	94% / 88%	71% / 62%	62% / 83%	83% / 76%	74% / 85%

For a fair comparison, we use two different datasets: (1) 132 (=106+26) patches used in the study of PATCH-SIM [56]—7 patches are excluded, since ODS results are missing for those 7 patches (confirmed by the ODS authors)—and (2) 325 (=242+83) patches, which are the intersection between the dataset available in Reference [60] and our extended dataset (Table 3).



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

Fig. 13. Comparison between Poracle and ODS. ODS tends to reject patches aggressively (see (a)), even rejecting many correct patches (see (b)).

RQ1: Our approach Poracle correctly rejects significantly more number of incorrect patches than PATCH-SIM (x1.9), Opad (x3.9), and BERT-LR (x1.5). Also, Poracle accepts almost all correct patches showing 99% precision, in contrast to ODS that rejects the largest number of correct patches among 5 considered approaches.

6.2.2 RQ2-1: Consistency. Given that Poracle uses a fuzzer, different results may be obtained at each fuzzing instance. PATCH-SIM shares the same issue due to its use of random test generation via Randoop [39]. To assess this concern, we run Poracle and PATCH-SIM 10 times for the PATCH-SIM dataset.[12] We use the same timeout as in the previous experiments (10 mins for Poracle and 35 mins for PATCH-SIM).

Table 7 shows the results. The "Rejected Consistently" column shows how consistently a patch is rejected, using notation X / Y where Y represents the number of patches that are rejected at least *once* out of 10 trials, while X represents the number of patches that are rejected in *all* 10 trials. The "Accepted Consistently" column similarly uses the X / Y notation. Note that PATCH-SIM—which uses a score-based approach—neither accepts nor rejects a patch when the timeout (35 mins) is reached or the tool crashes. We exclude those cases from the table. Our results show that Poracle returns *consistent* results across *all* 10 trials, in contrast to PATCH-SIM.

One possible explanation for our result is that an input causing a behavioral difference between patched and pre-patched versions is often nearby the original input. If that is the case, then a patch is likely to be rejected in a short amount of time during the fuzzing process.

[12]We use a smaller dataset to accommodate extended experimental time.

Table 7. Consistency of Patch Correctness Classification

Project	Rejected Consistently		Accepted Consistently	
	Poracle	PATCH-SIM	Poracle	PATCH-SIM
Chart	17 / 17	11 / 14	9 / 9	7 / 10
Lang	9 / 9	5 / 5	6 / 6	5 / 7
Math	42 / 42	19 / 25	41 / 41	44 / 57
Time	10 / 10	0 / 1	5 / 5	7 / 7
Total	78 / 78	35 / 45	61 / 61	63 / 81

Notation X / Y represents that Y patches are rejected/accepted at least once, and X patches are rejected/accepted in all 10 trials.

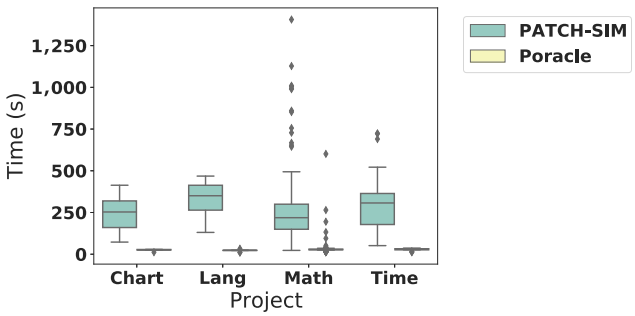


Fig. 14. Distribution of the execution time.

RQ2-1: Poracle showed consistent results in our experiments even though it utilizes a fuzzer.

6.2.3 RQ2-2: Time Efficiency of Fuzzing. We measure the time it takes for our fuzzer to reject a patch once started while the manual cost of writing a preservation condition is assessed with the user study described in Section 6.4.

The box-plot of Figure 14 shows the distribution of running time of Poracle and PATCH-SIM when both tools are run 10 times. The experiment is conducted on our extended dataset.[13] We compare Poracle with PATCH-SIM, because both approaches perform dynamic analysis by running the pre-patched and patched versions. We used the same setting of PATCH-SIM as described in Section 6.1. To avoid different classification results for each session, we used in all 10 sessions the same random tests Randoop generated. Recall that PATCH-SIM uses Randoop to prepare test cases. We measured the running time PATCH-SIM spent for patch classification and did not include the time taken to generate tests.

Meanwhile, comparison with Opad is not considered, since we simulate Opad using our custom fuzzer used in Poracle. The two ML-based approaches are not considered either. They do not involve running the program and finish almost instantly.

In the plot, X and Y axes show the subject projects and the duration (in sec), respectively. The overall average running time of Poracle is 32.57 s, which is about *8 times faster* than PATCH-SIM (257.26 s). This result again suggests that many incorrect patches can be easily rejected via fuzzing. This is in contrast to the earlier work using fuzzing [11, 59] where fuzzing is conducted for several hours. The key difference is that our approach can be applied at both the unit level and the system

[13]Timeout results of PATCH-SIM are excluded from the plot.

level, whereas in the earlier studies, fuzzing was conducted only at the system level. Note that the majority of the Defects4J tests are unit tests. We also note that the results shown in Figure 14 do not include the time spent writing preservation conditions. In Section 6.4, we show our user study results, including time information.

RQ2-2: The differential fuzzing module of Poracle *detected incorrect patches about 8 times faster than PATCH-SIM, the state-of-the-art dynamic patch classification technique.*

6.2.4 RQ2-3: Ablation Study on Preservation Conditions. To see the efficacy of preservation conditions, we conduct an ablation study by *removing preservation conditions* from our generalized tests. With these modified tests, all output differences are considered as evidence for patch incorrectness. Table 8 shows our results where the "Poracle" and "w/o P.C" column represent the results *with* and *without* preservation conditions, respectively. While about the same number of incorrect patches are *rejected correctly* (197 vs. 196), *a larger number of correct patches are rejected incorrectly when preservation conditions are removed (2 vs. 63).*

Not using a preservation condition can be simulated by using the preservation condition, "true." As described in Section 5.2.1, if an over-approximate preservation condition is used, then it may result in the rejection of a correct patch or the coincidental rejection of an incorrect patch based on wrong evidence. The latter happens when the observed output difference is not due to a regression error.

To check how often an incorrect patch is only coincidentally rejected, we perform the following: Given input I that causes an output difference between a pre-patched version P and its patched version P', we compute the output of the correct version P_c available in the benchmark. If P_c and P' produce the same output, then we consider that the patch was coincidentally rejected, because the observed output difference between P and P' does not indicate a regression error. In our ablation study, we found that 73 patches are only coincidentally rejected. In comparison, when preservation conditions are used, we observe no coincidental rejection.

RQ2-3: We summarize our findings as follows:

- *In a large portion of the patches (72%)—*$(196+63+73)/(326+132)$[a]*—output difference is observed between patched and pre-patched versions, showing the efficacy of using a parameterized test with fuzzing.*
- *However, only 60% of these inputs—*$196/(196+63+73)$*—accurately evidence the incorrectness of the patches.*
- *By using preservation conditions, those wrong evidences for rejection are filtered out.*
- *These results suggest that many incorrect patches can be correctly filtered out* via the combination of fuzzing and preservation conditions.

[a]This ratio is computed with the results from the extended dataset. Similar results are obtained from the PATCH-SIM dataset. Recall that the extended dataset is the superset of the PATCH-SIM dataset.

6.2.5 RQ2-4: Application Scope. To assess the performance with wider ranges of patches, we use the 77 developer-written correct patches for the 77 buggy versions in our dataset, which tend to be longer than the tool-generated patches; the tool-generated patches in our dataset use, on average, 2.81 lines and 14.15 words, whereas the developer patches use, on average, 5.83 lines and 29.29 words. Table 9 shows the result. Poracle shows *almost perfect* performance, rejecting only one correct patch for the Math-71 bug in Defects4J.[14] As described in Section 5.2.1, our method can reject a correct patch if the preservation condition is over-approximate. For the Math-71 bug,

[14]ODS also rejects the same correct patch.

Table 8. Comparison between (1) Poracle and (2) Poracle without Preservation Conditions (column "w/o P.C")

Project	Patches		Rejected Correctly		Rejected Incorrectly		Rejected Coincidentally	
	Incorrect	Correct	Poracle	w/o P.C	Poracle	w/o P.C	Poracle	w/o P.C
Chart	24 / 24	2 / 2	17 / 17	19 / 19	0 / 0	1 / 1	0 / 0	3 / 3
Lang	11 / 32	4 / 19	9 / 19	7 / 14	0 / 0	3 / 16	0 / 0	2 / 11
Math	64 / 250	19 / 98	42 / 148	45 / 148	0 / 2	7 / 46	0 / 0	12 / 58
Time	13 / 20	2 / 13	10 / 13	10 / 15	0 / 0	0 / 0	0 / 0	1 / 1
Total	112 / 326	27 / 132	78 / 197	81 / 196	0 / 2	10 / 63	0 / 0	18 / 73

In notation x/y, x and y represent the data for the first dataset and the data for the extended dataset, respectively.

Table 9. Results for Developer Patches

Project	Patches	Erroneously Rejected				
		Poracle	Opad	PATCH-SIM	ODS	BERT-LR
Chart	13	0	1	3	3	7
Lang	10	0	2	3	4	3
Math	45	1	6	8	17	14
Time	9	0	0	2	4	6
Total	77	1	9	16	28	30

In each row, a better-performing tool is highlighted.

we used the assertion used in the original test as a preservation condition, which turns out to be over-approximate.

Meanwhile, the other four tools incorrectly reject a higher number of correct patches. The result of the two ML-based techniques (ODS and BERT-LR) are particularly alarming—about 36% and 39% of the patches are erroneously rejected in ODS and BERT-LR, respectively.

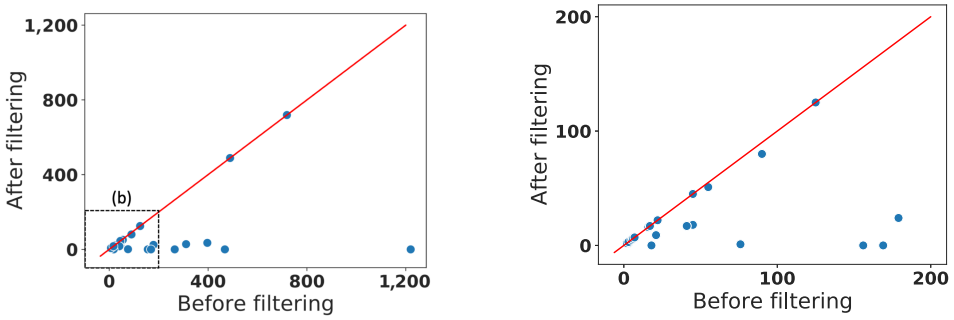
> *RQ2-4: Poracle almost always does not reject human-written correct patches, which indicates that the tool is not overfitting to auto-generated patches. In comparison, the existing approaches often misjudge human-written patches and reject them erroneously.*

6.3 RQ3: Assessing Cost Reduction

As mentioned in Section 1, recent APR systems [4, 11, 12, 53] return a list of plausible patches. Sifting out a correct patch from a large number of incorrect patches is a costly process. PC techniques such as Poracle can help reduce costs by filtering out incorrect patches from the list. We conduct a separate experiment to assess the cost reduction achieved by Poracle.

6.3.1 Experimental Settings. To estimate cost reduction with an actual APR system, we collect patches from JAID [4], a state-of-the-art APR tool that returns a ranked list of patches. We apply JAID to the 77 Defects4J buggy versions for which we have generalized tests and obtain the ranked lists of plausible patches from 28 versions. For the remaining versions, JAID fails to generate plausible patches.

In this experiment, we measure how many patches should be reviewed *before* and *after* applying Poracle (with 10-min timeout), assuming that patches are reviewed in the order of their rankings.



(a) Overall results

(b) Results for the patches whose original rankings are below 200

Fig. 15. The number of patches to review *before* (X-axis) and *after* (Y-axis) applying Poracle.

Before applying Poracle, the number of patches to review is identical to the ranking of the correct patch, if one exists in the list. If there is no correct patch in the list, then we report the total number of patches in the list. Please note that to determine whether a patch is correct or not, we use the correctness labels provided by the authors of JAID.[15]

Once Poracle is applied, a correct patch that was originally ranked at the nth position is re-ranked to the $(n - m)$-th position if (1) m incorrect patches that were ranked higher than the correct patch are filtered out by Poracle and (2) the correct patch is not filtered out. If the ranked list does not contain a correct patch, then we report the number of patches that remain after applying Poracle, since the users, without knowing whether there exists a correct patch in the list, will need to review the remaining patches.

6.3.2 Experimental Results. Figure 15(a) shows our results. In the figure, each data point denotes a bug in the Defects4J benchmark [19]. The X and Y axes represent the number of patches to review *before* and *after* applying Poracle, respectively, until either the correct patch is found or all patches are reviewed and no correct patch is found. Figure 15(b) shows a closer look at the results for the patches whose original X-values are below 200. Note that all data points are located under the red diagonal line, indicating that correct patches are detected either earlier or at the same time as they would be without employing our method. In most cases (20 out of 28), the number of patches to review decreases (i.e., data points are close to the X-axis). For example, in Chart9, a correct patch originally ranked at 45th is ranked up to 18th after applying Poracle. Also, in Math28, all 1,220 incorrect patches in the list are filtered out. Overall, our approach reduces the number of patches to review by 108 patches per version, resulting in a reduction ratio of 39.83%.

We also processed the same JAID patches with PATCH-SIM using the setting described in Section 6.1. In our experiment, PATCH-SIM filters out incorrect patches more aggressively than Poracle, resulting in a reduction ratio of 70.6%. F-measure is also higher with PATCH-SIM at 91% compared to 79% with Poracle. It is important to note, however, that PATCH-SIM also filters out 12 correct patches out of 14 generated by JAID, whereas Poracle retains all correct patches. Our results suggest that by using Poracle, developers can reduce the number of patches to review while retaining all correct patches, which is difficult to be achieved with PATCH-SIM or other score-based approaches.

[15] https://bitbucket.org/maxpei/jaid/wiki/Home

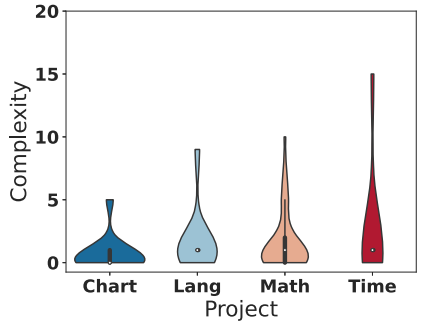


Fig. 16. The violin plot showing the complexities of our preservation conditions.

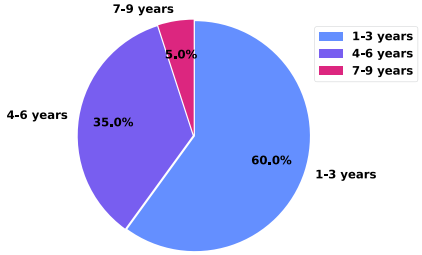


Fig. 17. The number of years our participants had been programming.

> *RQ3: PORACLE reduces the number of patches to review by 39.83% per version while retaining all correct patches.*

6.4 RQ4: Assessing Usability

Our experience suggests that it is straightforward to generalize existing failing tests with preservation conditions. Figure 16 shows the complexities of the 77 preservation conditions we wrote.[16] We estimate the complexities of the preservation conditions by counting the number of operators (e.g.,>, >=, && and ||) used in a preservation condition. As shown in Figure 16, the preservation conditions we wrote are quite simple, using, on average, only 1.58 operators. To assess the usability of our approach more objectively, we conduct a user study.

6.4.1 User Study Setup. We conducted a user study with 66 junior/senior undergraduate students in a third-year course (Software Engineering) at Ulsan National Institute of Science and Technology in 2022. Figure 17 displays the number of years our participants had been programming. To motivate the participants, grade compensation was provided.

To conduct the user study, we prepared four questions, each consisting of a buggy version, a failing test, and 10 patches. We extracted all buggy versions from the Defects4J benchmark [19]; see the "Bug ID" column of Table 10. We used the following criteria to select the buggy versions and the patches:

— Each buggy version requires a different pattern of preservation conditions; see the "Pattern" column of Table 10.

[16]We trim the tails of the violin plots that are beyond the range of the data. Note that a violin plot is often stretched (smoothed) out beyond the range of the data.

Table 10. Four Questions Used in the User Study

Question	Group		Bug ID	Complexity	Pattern	Pattern Complexity
	Group 1	Group 2				
Q1	Poracle	Manual	Math-73	10	CC (Complementary Cases)	2.75
Q2	Poracle	Manual	Math-105	1	EGA (Existing Assertion)	2.24
Q3	Manual	Poracle	Math-28	0	UE (Unexpected Exception)	0
Q4	Manual	Poracle	Lang-58	1	RI (Reference Implementation)	2.08

— Our method is designed for developers having domain-specific knowledge of the buggy version. To meet this assumption, we selected three buggy versions extracted from the Apache Commons Math project[17] and one buggy version extracted from Apache Commons Lang project.[18] These two projects implement common operations for math and string manipulation with which the participants are likely to be familiar.
— All patches should pass the failing test.
— For each version, we used the ground-truth patch available in the Defects4J benchmark as a correct patch.
— For each version, we randomly selected nine incorrect patches from those Poracle successfully filtered out in our experiments presented in Section 6.2.

In Table 10, the "Complexity" column shows the complexities of the ground-truth preservation conditions for each question. Compare them with the "Pattern Complexity" column showing the mean complexity of the preservation conditions for each pattern; we collect all preservation conditions for each pattern and then compute the mean complexity of them.

We compare the user experience of patch assessment between the following two patch assessment methods:

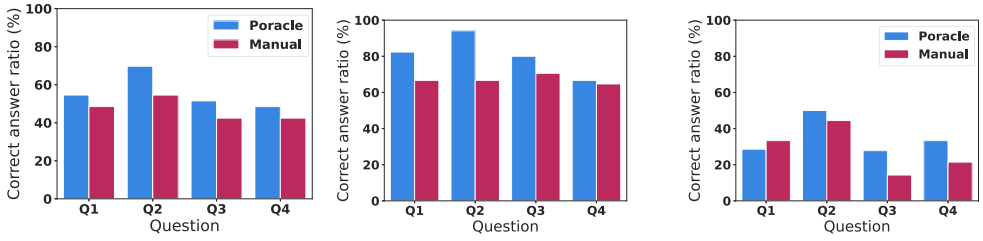
(1) Manual patch assessment: The user identifies a correct patch out of 10 given patches consisting of 1 correct patch and 9 incorrect patches. Poracle is not used in this case.
(2) Semi-automatic patch assessment using Poracle: The user extends a failing test with a preservation condition and runs Poracle to identify a correct patch out of the same 10 patches as used in manual patch assessment. To make sure the participants do not manually find a correct patch, we instructed them to submit the result they obtained using Poracle.

We do not include fully automated patch classification techniques mentioned in Section 6.2, because those methods do not change the manual patch assessment process—i.e., given a list of patches ranked top 10, the developer looks for a correct patch. In both assessment methods (manual and ours), we provided the participants with the necessary domain knowledge to write correct answers, such as API documents. We compare the correct answer ratio between the experimental group (in which Poracle is used) and the control group (in which manual assessment is used).

Since our user-study participants are not familiar with APR and preservation conditions, we provided a single session of a 75-minute tutorial on APR, preservation conditions, and how to use Poracle. We did not provide the guideline on how to write preservation conditions described in Section 5.3. We will discuss how this affects the results in Section 6.4.3.

We provided the user with a docker container where Poracle is installed. The container also contains the materials for the four questions, where each question consists of the following materials.

[17] https://commons.apache.org/proper/commons-math/
[18] https://commons.apache.org/proper/commons-lang/



(a) All students (b) Top 50% students (c) Last 50% students

Fig. 18. Correct answer ratios.

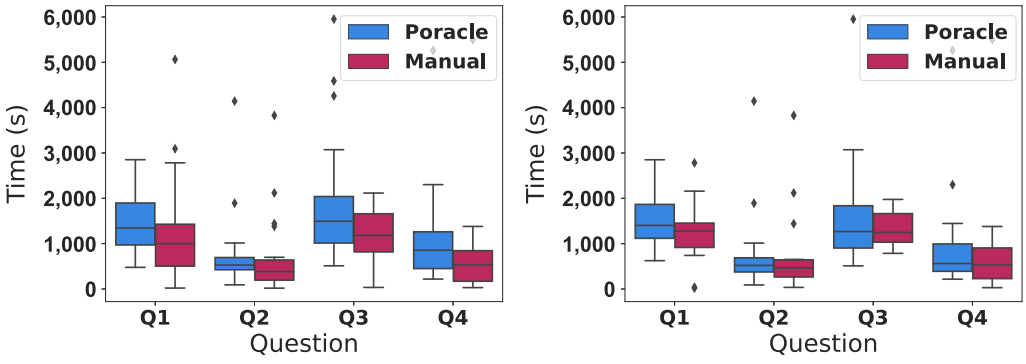
— Source code for the buggy version
— A failing test for the buggy version
— Editors including Vim, Emacs, and Nano
— A script to run the failing test
— 10 patch candidates, all of which pass the given failing test
— A script to run Poracle

Only the last one is used exclusively in the experimental group. The remaining materials are used in both groups.

We randomly divided the 66 participants into two groups and assigned each group four questions—two for manual patch assessment and two for semi-automatic patch assessment. The "Group" column of Table 10 shows how we distribute the four questions into the two groups. For each question, we prepared two versions—one for the control group and the other one for the experimental group. Both question versions are extracted from the same bug of Defects4J [19], as shown in the "Bug ID" column. For each question, we distributed its two versions into the two groups. For example, for Q1, we prepared two versions of the question extracted Math-73 and assigned the semi-automatic assessment version to the first group and the manual assessment version to the second group. This study design is to prevent the result of one version of a question Q (say, the semi-automatic assessment version of Q) from being affected by another version of the same question (the manual assessment version of Q). The four questions we used in the study cover all four patterns of preservation invariants described in Section 4.3, as shown in the "Pattern" column.

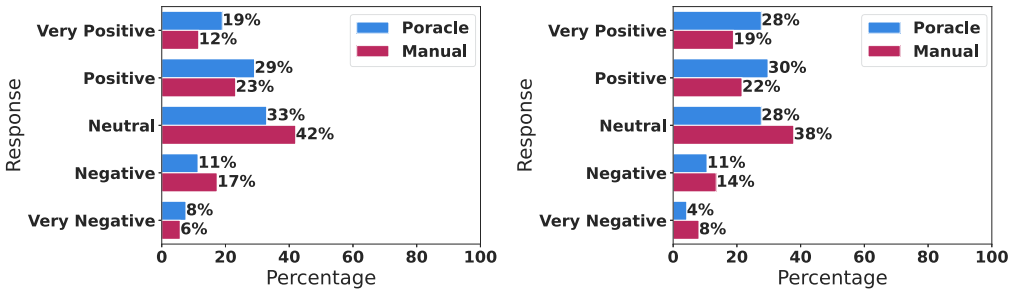
6.4.2 User Study Results. As shown in Figure 18(a), the correct answer ratio is higher in the experimental group where Poracle is used than in the control group where patches are manually assessed. When evaluating the participants' answers, we gave either 1 point (if an answer is correct) or 0 point (if an answer is wrong). While the result suggests that using Poracle can help the users find correct patches by filtering out incorrect ones, the overall correctness ratios are not very high. However, Figures 18(b) and 18(c) provide different perspectives. After the semester was over, we split the 66 students into two groups, the top 50% students and the last 50% students, based on their total scores accumulated throughout the semester (e.g., exam scores and assignment points), except for the compensation points assigned for the user study. The correctness ratios are clearly higher in the top-50% group than in the last-50% group, suggesting that high-performing participants make better use of Poracle. In addition, the score differences between the two methods (Poracle and manual) are statistically significant (p-value < 0.05; we have conducted the Mann-Whitney rank test [30]).

Meanwhile, the box plots in Figure 19 compare the manual time cost involved in each method—i.e., the time taken to write preservation conditions (Poracle) and the time to review 10 patches



(a) All students (b) Students who submitted correct answers

Fig. 19. Manual time cost.



(a) All students (b) Top 50% students

Fig. 20. Experiences about the two patch assessment methods.

(Manual). When considering all students, the manual time cost of writing a preservation condition is higher than that of the manual method (see Figure 19(a)). However, if we consider only those who submitted correct answers (they better represent the groups who used each method effectively), the gap between the two methods is only marginal, as shown in Figure 19(b); the median times between the two methods are almost the same, and there is no statistically significant difference between the two methods (p-value < 0.05 from the Mann-Whitney rank test).

Apart from the correctness ratio, we also asked the participants about their experiences in using the two patch assessment methods they used. As shown in Figure 20(a), more participants expressed a positive experience with using Poracle than with the manual method (48% vs. 35%). The high-performing group responded similarly (58% vs. 41%), as shown in Figure 20(b). In both figures, the response differences between the two methods (Poracle and manual) are statistically significant (p-value < 0.05 from the Mann-Whitney rank test).

Last, we asked the participants to choose a preferred patch assessment method between the two methods they used. Most participants chose the semi-automatic assessment method using Poracle over the manual one, as shown in Figure 21.

6.4.3 Discussion about the User Study Results. We believe that our method is cost-effective for several reasons. Once a preservation condition is written, it can be reused to filter out any number of incorrect patches fixing the same bug. As shown in Section 6.3, the one-time cost of writing

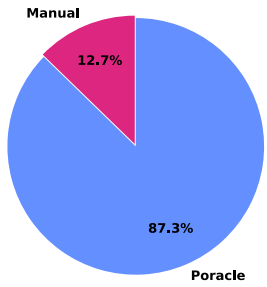


Fig. 21. Poracle vs. Manual.

Table 11. Distributions of Correct, Over-approximate, Under-approximate, and Wrong p.c. (Preservation Conditions) Written by the Participants

Question	Ground-truth Pattern	Correct p.c.	Over-approximate p.c.	Under-approximate p.c.	Wrong p.c.
Q1	CC	21	2	5	5
Q2	EGA	25	0	1	7
Q3	UE	18	0	9	6
Q4	RI	17	1	7	8

Table 12. Frequency of (in)Correct Patches Submitted by the Participants for Each Category of p.c. (Preservation Condition) Described in Table 11

Question	Ground-truth Pattern	Correct p.c.	Over-approximate p.c.		Under-approximate p.c.		Wrong p.c.	
		Correct	Correct	Incorrect	Correct	Incorrect	Correct	Incorrect
Q1	CC	21	0	2	0	5	0	5
Q2	EGA	25	0	0	0	1	0	7
Q3	UE	18	0	0	0	9	0	6
Q4	RI	17	0	1	0	7	0	8

preservation condition can be compensated by filtering out tens and hundreds of incorrect patches. In addition, when a regression error occurs in the future, preservation conditions can be reused to filter out incorrect patches for that error.

Our approach is intended for developers who want more control over the patch classification process. Indeed, our user study results suggest that our approach is more likely to be useful for advanced developers. Top-50% students used our approach more effectively, and their opinions on our approach were more positive than those of bottom-50% students.

Similar to other specification-based approaches, the effectiveness of our approach depends on the quality of the preservation conditions the users wrote. Table 11 displays the distributions of correct, over-approximate, under-approximate, and wrong preservation conditions written by the participants. Please refer to Section 5.2.1 for the definitions of over-approximate, under-approximate, and wrong preservation conditions.[19] We compared the preservation conditions written by the participants with the ground-truth preservation conditions used in our tool experiments (Section 6.2). As demonstrated in Table 12, participants in our study submitted incorrect answers (i.e., failed to identify the correct patch) when they used incorrect (i.e., over-approximate, under-approximate, or wrong) preservation conditions. When an under-approximate preservation condition is used, 10 participants identified multiple patches, including the ground-truth patch, as correct, instead

[19]Uncompilable conditions are considered as "wrong."

Table 13. Distributions of Participants' Incorrect Answers and Incorrect Patterns

Question	Ground-truth Pattern	Incorrect Answer	Incorrect Pattern
Q1	CC	12	7
Q2	EGA	8	7
Q3	UE	15	15
Q4	RI	16	11

of selecting just one. This can happen because an under-approximate preservation condition may fail to filter out all incorrect patches (see Figure 9(a)).

In Section 5.3, we discussed the guidelines for writing preservation conditions. To assess the effectiveness of the guideline had we provided it, we collected the participants' preservation conditions that led to incorrect answers and analyzed how often they used the same preservation condition patterns as those used in the ground-truth ones. As shown in Table 13, our participants often used incorrect patterns different from those used in the ground-truth preservation conditions when they failed to identify the correct patch. We hypothesize that providing the guidelines could help developers in writing correct preservation conditions, which should be investigated in future work.

> *RQ4: Our user-study participants were more successful in finding correct patches when using* Poracle *as compared to when they did not use it. And a larger number of participants reported a positive experience with using Poracle compared to the manual approach.*

7 THREATS TO VALIDITY

External Validity: Our findings might not be valid for patches that are not in our dataset. Also, the distribution of correct and incorrect patches may be different, depending on an APR tool used to generate the patches. To mitigate this threat, we conducted experiments with an extended dataset in addition to the PATCH-SIM dataset and obtained similar results from both datasets.

For the 77 buggy versions in our dataset, we could successfully write preservation conditions using the four patterns (i.e., UE, CC, EGA, and RI). However, there could be cases that cannot be covered with these four patterns, and we do not claim that they are exhaustive. Instead, we would like to point out that our approach is designed to be generic. Developers can apply our approach as long as they can express preservation conditions using the preserveIf and failToPreserve methods we provide.

Regarding the user study, our findings may not be generalized to all programmers. However, we note that our participants are junior/senior students majoring in computer science who can be considered entry-level developers. When writing a preservation condition, different levels of familiarity of participants with the subject code may confound the result. To mitigate this threat, the code in our survey questions deals with mathematical computation (Math-73, Math-105, and Math-28) or string manipulation (Lang-58) to which our participants are likely to have similar levels of understanding.

Internal Validity: To decide whether the classification result is correct, we use the labels prepared manually by other researchers in previous works [27, 56], which is subject to bias. Also, generalized tests prepared by us are subject to bias, though the existing failing tests clearly reveal bug fixing intention for most bugs in our dataset. To mitigate the threat posed by using manual labels and specifications, we validate all rejection decisions with ground-truth versions available in the

Defects4J benchmark. Specifically, given an obtained input I that causes a behavioral difference between patched and pre-patched versions, we run the correct version with the same input I. If the correct version and the patched version produce different outputs for I, then the patch is indeed incorrect. Through this process, *we identified four misclassified patches* and used rectified labels in the experiments.

In the user study, we provided the participants with 10 patches for each buggy version. However, the number of patches may vary, depending on the APR tool used to generate the patches, and this may affect the result of the user study. In this work, we treat the number of patches as a constant and do not consider the effect of the number of patches on the result.

Our user study was conducted assuming the developers have domain-specific knowledge about their software project. To fulfill this assumption, we provided the study participants with materials such as API documents from which they can extract domain-specific knowledge, as described in Section 6.4. However, our study was not conducted with the original developers of the subject software projects, and whether developers have sufficient knowledge to write a preservation condition should be investigated in a separate study. Nevertheless, our experience with writing preservation conditions suggests that the developers are likely to have sufficient domain-specific knowledge. Although we are not the original developers, we could write preservation conditions based on API documents and bug reports.

In the user study, our participants used traditional editors such as Vim, and we did not provide an **IDE (Integrated Development Environment)**. Using an IDE could have affected the participants' performance in both experimental and control groups, since an IDE provides various useful features such as code completion and a debugger. However, we chose not to provide an IDE, because our participants are more accustomed to using Vim than an IDE, since Vim is the primary editor used in most computer science courses they took.

8 CONCLUSION AND FUTURE WORK

In this work, we have explored the possibility of using a semi-automated approach for APR to mitigate its overfitting problem. Essentially, our approach automatically generates hold-out tests based on a snippet of user information (i.e., a preservation condition). Our positive results suggest that there is room for research on semi-automated approaches for APR. Given that the current APR systems often generate incorrect patches and currently there is no automatic system to check the patch correctness reliably (current automatic patch classification techniques often make a wrong decision, as shown in Section 6.2), semi-automatic approaches like ours can be a practical solution to the overfitting problem of APR.

One potential future research direction is to find out an optimal way to interact with the user. An ideal method will cause the minimum cognitive load to the user. Given that most APR systems use tests written by the developers, utilizing those tests as done in this study can be one of the promising directions. While our user-study results are supportive of our semi-automatic approach, a more extensive user study is needed to obtain a deeper understanding of how developers perceive our approach, which we leave as future work.

REFERENCES

[1] Michael Buckland and Fredric Gey. 1994. The relationship between recall and precision. *J. Amer. Soc. Inf. Sci.* 45, 1 (1994), 12–19.

[2] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolo Perino, and Mauro Pezze. 2013. Automatic recovery from runtime failures. In *35th International Conference on Software Engineering (ICSE'13)*. IEEE, 782–791.

[3] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. 2010. Automatic workarounds for web applications. In *18th 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 Trans. Softw. Eng.* 47, 12 (2020).
[5] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A scalable tree boosting system. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD'16)*. 785–794.
[6] Tsong Yueh Chen, D. H. Huang, T. H. Tse, and Zhi Quan Zhou. 2004. Case studies on the selection of useful relations in metamorphic testing. In *4th Ibero-American Symposium on Software Engineering and Knowledge Engineering ('JIISIC'04)*. Citeseer, 569–583.
[7] Koen Claessen and John Hughes. 2000. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP'00)*. 268–279.
[8] 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.
[9] The Apache Software Foundation. 2023. The API document of the GCD method. Retrieved from https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/util/ArithmeticUtils.html#gcd(int,%20int)
[10] The Apache Software Foundation. 2023. The API document of the inverseCumulativeProbability method. Retrieved from https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/distribution/NormalDistribution.html#inverseCumulativeProbability(double)
[11] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In *28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 8–18.
[12] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 19–30.
[13] 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.
[14] Paul Holser. 2014. junit-quickcheck: Property-based testing, JUnit-style. Retrieved from https://pholser.github.io/junit-quickcheck/
[15] JetBrains. 2000. IntelliJ IDEA. Retrieved from https://www.jetbrains.com/idea/
[16] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *International Symposium on Software Testing and Analysis (ISSTA'18)*. ACM, 298–309.
[17] Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equivalent code fragments via random testing. In *18th International Symposium on Software Testing and Analysis*. 81–92.
[18] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *45th International Conference on Software Engineering*. 1430–1442.
[19] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis (ISSTA'14)*. 437–440.
[20] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering (ICSE'13)*. 802–811.
[21] Hyungsub Kim, Muslum Ozgur Ozmen, Z. Berkay Celik, Antonio Bianchi, and Dongyan Xu. 2023. PatchVerif: Discovering faulty patches in robotic vehicles. In *USENIX Security Symposium*.
[22] YoungJae Kim, Seungheon Han, Askar Yeltayuly Khamit, and Jooyong Yi. 2023. Automated program repair from fuzzing perspective. In *32nd International Symposium on Software Testing and Analysis (ISSTA'23)*. Association for Computing Machinery, New York, NY, 854–866. DOI: https://doi.org/10.1145/3597926.3598101
[23] Tien-Duy B. Le, Jooyong Yi, David Lo, Ferdian Thung, and Abhik Roychoudhury. 2014. Dynamic inference of change contracts. In *IEEE International Conference on Software Maintenance and Evolution*. IEEE, 451–455.
[24] C. Le Goues, ThanhVu Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.* 38, 1 (Jan. 2012), 54–72.
[25] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *31st IEEE/ACM International Conference on Automated Software Engineering*. 602–613.
[26] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *International Symposium on Software Testing and Analysis (ISSTA'19)*. 31–42.
[27] 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'20)*. 615–627.

[28] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'16)*. 298–312.
[29] Fan Long and Martin C. Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *International Conference on Software Engineering (ICSE'16)*. 702–713.
[30] Henry B. Mann and Donald R. Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Stat.* 18, 1 (1947), 50–60.
[31] Johannes Mayer and Ralph Guderlei. 2006. An empirical study on the selection of good metamorphic relations. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, Vol. 1. IEEE, 475–484.
[32] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for simple program repairs. In *International Conference on Software Engineering (ICSE'15)*. 448–458.
[33] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE'16)*. 691–701.
[34] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. 2013. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 65–74.
[35] Microsoft. 2021. Visual Studio Code. Retrieved from https://code.visualstudio.com/
[36] Martin Monperrus. 2018. Automatic software repair: A bibliography. *ACM Comput. Surv.* 51, 1 (2018), 1–24.
[37] 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'21)*. IEEE, 229–240.
[38] 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'20)*. IEEE, 1273–1285.
[39] 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.
[40] 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'19)*. 398–401.
[41] 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'16)*. 1181–1192.
[42] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis (ISSTA'11)*. 199–209.
[43] 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'17)*. IEEE, 615–632.
[44] Arooba Shahoor, Askar Yeltayuly Khamit, Jooyong Yi, and Dongsun Kim. 2023. LeakPair: Proactive repairing of memory leaks in single page web applications. In *38th International Conference on Automated Software Engineering (ASE'23)*.
[45] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic program repair. In *42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 390–405.
[46] 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 *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*. 532–543.
[47] 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'16)*. 727–738.
[48] Haoye Tian, Yinghua Li, Weiguo Pian, Abdoul Kader Kabore, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé. 2022. Predicting patch correctness based on the similarity of failing test cases. *ACM Trans. Softw. Eng. Methodol.* 31, 4 (2022), 1–30.
[49] 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'20)*. IEEE, 981–992.
[50] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests. *ACM SIGSOFT Softw. Eng. Notes* 30, 5 (2005), 253–262.
[51] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated patch correctness assessment: How far are we? In *35th IEEE/ACM International Conference on Automated Software Engineering*. 968–980.
[52] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *International Conference on Software Engineering (ICSE'18)*. ACM, 1–11.
[53] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: Balancing edit expressiveness and search effectiveness in automated program repair. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. 354–366.

[54] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: Revisiting automated program repair via zero-shot learning. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 959–971.
[55] 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'17)*. 226–236.
[56] 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'18)*. 789–799.
[57] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *International Conference on Software Engineering (ICSE'17)*. 416–426.
[58] Bo Yang and Jinqiu Yang. 2020. Exploring the differences between plausible and correct patches at fine-grained level. In *IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF'20)*. IEEE, 1–8.
[59] 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'17)*. 831–841.
[60] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. Automated classification of overfitting patches with statically extracted code features. *IEEE Trans. Softw. Eng.* 48, 8 (2021).
[61] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the QuixBugs benchmark. *J. Syst. Softw.* 171 (2021), 110825.
[62] Jooyong Yi and Elkhan Ismayilzada. 2022. Speeding up constraint-based program repair using a search-based technique. *Inf. Softw. Technol.* 146 (2022), 106865.
[63] Jooyong Yi, Dawei Qi, Shin Hwei Tan, and Abhik Roychoudhury. 2013. Expressing and checking intended changes via software change contracts. In *International Symposium on Software Testing and Analysis*. 1–11.
[64] Jooyong Yi, Dawei Qi, Shin Hwei Tan, and Abhik Roychoudhury. 2015. Software change contracts. *ACM Trans. Softw. Eng. Methodol.* 24, 3 (2015), 1–43.
[65] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2019. Alleviating patch overfitting with automatic test generation: A study of feasibility and effectiveness for the Nopol repair system. *Empir. Softw. Eng.* 24, 1 (2019), 33–67.

Received 24 January 2023; revised 30 July 2023; accepted 6 September 2023