2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

On Reliability of Patch Correctness Assessment

Xuan-Bach D. Le[1], Lingfeng Bao[2], David Lo[3], Xin Xia[4], Shanping Li[5], and Corina Pasareanu[1,6]

[1] Carnegie Mellon University, USA, {bach.le,corina.pasareanu}@west.cmu.edu
[2] Zhejiang University City College, China, lingfengbao@zju.edu.cn
[3] Singapore Management University, Singapore, davidlo@smu.edu.sg
[4] Monash University, Australia, xin.xia@monash.edu
[5] Zhejiang University, China, shan@zju.edu.cn
[6] NASA Ames Research Center, USA, Corina.S.Pasareanu@nasa.gov

***Abstract*—Current state-of-the-art automatic software repair (ASR) techniques rely heavily on incomplete specifications, or test suites, to generate repairs. This, however, may cause ASR tools to generate repairs that are incorrect and hard to generalize. To assess patch correctness, researchers have been following two methods separately: (1) Automated annotation, wherein patches are automatically labeled by an independent test suite (ITS) – a patch passing the ITS is regarded as correct or generalizable, and incorrect otherwise, (2) Author annotation, wherein authors of ASR techniques manually annotate the correctness labels of patches generated by their and competing tools. While automated annotation cannot ascertain that a patch is actually correct, author annotation is prone to subjectivity. This concern has caused an on-going debate on the appropriate ways to assess the effectiveness of numerous ASR techniques proposed recently.**

In this work, we propose to assess reliability of author and automated annotations on patch correctness assessment. We do this by first constructing a gold set of correctness labels for 189 randomly selected patches generated by 8 state-of-the-art ASR techniques through a user study involving 35 professional developers as independent annotators. By measuring inter-rater agreement as a proxy for annotation quality – as commonly done in the literature – we demonstrate that our constructed gold set is on par with other high-quality gold sets. We then compare labels generated by author and automated annotations with this gold set to assess reliability of the patch assessment methodologies. We subsequently report several findings and highlight implications for future studies.

I. Introduction

Bug fixing is notoriously difficult, time-consuming, and costly [1], [2]. Hence, effective automatic software repair (ASR) techniques that can help reduce the onerous burden of this task, would be of tremendous value. Interest in ASR has intensified in recent years as demonstrated by substantial work devoted to the area [3]–[14], bringing the futuristic idea of ASR closer to reality. ASR can be divided into two main families: heuristics- vs. semantics-based approaches, based on the way they generate and traverse the search space for repairs.

Ideally, complete specifications should be used for assessing correctness of patches generated by ASR. It is, however, very hard to obtain complete specifications in practice. ASR techniques thus typically resort to using test cases as the primary criteria for correctness judgment of machine-generated patches – a patch is considered *correct* if it passes all the tests used for repair [9]. This assessment methodology, however, has been shown to be ineffective. There could be multiple patches that pass all the tests but are still *incorrect* [15], [16], causing the so-called patch overfitting [17], [18]. This happens because the search space is often very large and contains many plausible repairs, which unduly pass all tests but fail to generalize. This thus motivates the need of new methodologies to assess patch correctness. The new methodologies need to rely on additional criteria instead of using the test suite used for generating repair candidates (aka. *repair test suite*) alone.

To address this concern, recent works have been following two methods for patch correctness assessment separately:

- **Automated annotation by independent test suite.** Independent test suites obtained via an automatic test case generation tool are used to determine correctness label of a patch – see for example [17], [19]. Following this method, a patch is deemed as *correct* or *generalizable* if it passes both the repair and independent test suites, and *incorrect* otherwise.
- **Author annotation.** Authors of ASR techniques manually check correctness labels of patches generated by their own and competing tools – see for example [20], [21]. Following this method, a patch is deemed as *correct* if authors perceive a semantic equivalence between the generated patches and the original developer patches.

While the former is incomplete, in the sense that it fails to prove that a patch is actually correct, the latter is prone to author bias. In fact, these inherent disadvantages of the methods have caused an on-going debate in the program repair community as to which method is better for assessing the effectiveness of various ASR techniques being proposed recently. Unfortunately, there has been no extensive study that objectively assesses the two patch validation methods and provides insights into how the evaluation of ASR's effectiveness should be conducted in the future.

In this work, we conduct a study that addresses this gap in research. We start by creating a gold set of correctness labels for a collection of ASR generated patches, and subsequently use it to assess reliability of labels created through author and automated annotations. We study a total of 189 patches generated by 8 popular ASR techniques (ACS [20], Kali [15], GenProg [20], Nopol [8], S3 [22], Angelix [4], and Enumera-

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00064

tive and CVC4 embedded in JFix [13]). These patches are for buggy versions of 13 real-world projects, of which six projects are from Defects4J [23] (Math, Lang, Chart, Closure, Mockito, and Time) and seven projects are from S3's dataset [22] (JFlex, Fyodor, Natty, Molgenis, RTree, SimpleFlatMapper, Graph-Hoper). To determine correctness of each patch, we follow best practice by involving multiple independent annotators in a user study. Our user study involves 35 professional developers; each ASR-generated patch is labeled by five developers by comparing the patch with its corresponding ground truth patch created by the original developer(s) who fixed the bug. We then analyze the reliability of created gold set and compare it with labels generated by three groups of ASR tool authors [21], [22], [24] and two automatic test case generation tools such as DIFFTGEN that has been used in prior study [25] and RANDOOP [26] that we use in this study. We answer three research questions:

RQ1 *Can independent annotators agree on patch correctness?*
RQ2 *How reliable are patch correctness labels generated by author annotation?*
RQ3 *How reliable are patch correctness labels inferred through automatically generated independent test suite?*

In RQ1, by measuring inter-rater agreement as a proxy of annotation quality – as commonly done in the literature [27], [28] – we demonstrate that our gold set has substantial inter-rater scores and thus is on par with other high-quality gold sets. In the subsequent two RQs, we investigate the strengths and deficiencies of author and automated patch correctness annotation.

We summarize our contributions below:

- We are the first to investigate the reliability of author and automated annotation for assessing patch correctness. To perform such assessment, we have created a gold set of labelled patches created by a user study involving 35 professional developers. By means of this gold set, we highlight strengths and deficiencies in popular assessment methods employed by existing ASR studies.
- Based on the implications of our findings, we provide several recommendations for future ASR studies to better deal with patch correctness validation. Specifically, we find that automated annotation, despite being less effective as compared to author annotation, can be used to augment author annotation and reduce the cost of manual patch correctness assessment.

The rest of the paper is organized as follows. Section II describes background for this work. Section III describes how we collect gold set of patch correctness labels. We answer RQs to assess the quality of our gold set, author annotation, and automated annotation in Section IV, V, and VI respectively. Section VII discusses our findings, post-study survey, threats to validity, and future extensions. Section IX concludes.

II. BACKGROUND

In this section, we describe automated software repair (ASR) techniques used in our experiments. We subsequently describe popular patch validation methods used in ASR research. Finally, we discuss best practices in building gold sets.

ASR techniques: GenProg [9] is one of the first techniques that sparked interests in ASR. Given a buggy program and a set of test cases, at least one of which is failing, GenProg uses a number of mutation operators, such as statement delete, insert, and append, to create a large pool of repair candidates. It then uses genetic programming to apply the mutations and evolve the buggy program until a candidate passing all the tests is found. Kali [15] is a naive ASR technique, which just blindly deletes any statements that are identified as potentially buggy. Despite being very simple, Kali has been shown to be as effective and efficient as GenProg. Nopol [8] is a recently developed ASR technique that focuses on only repairing defective *if-conditions*. Nopol attempts to synthesize an if-condition expression that renders all the tests to pass by using program synthesis. In a similar vein, ACS [20] also focuses on synthesizing repairs for buggy if-conditions. Like Nopol, ACS also uses program synthesis to create repairs. Unlike Nopol, ACS attempts to rank the repair candidates using various ranking functions. Angelix [4], S3 [22], and JFix [13] use symbolic execution and constraint solving to infer specifications and various program synthesis techniques to synthesize repairs conforming to the inferred specifications. Angelix uses component-based synthesis [29], while S3 and JFix use syntax-guided synthesis [30].

Evaluation of ASR Generated Patches: Initially in ASR research, test cases were used as the sole criteria for judging correctness of machine-generated patches. By relying on the assumption that a patch that passes the repair test suite is regarded as correct, early repair techniques such as GenProg [9], AE [31], and RSRepair [32] reported to produce many such correct patches. However, it has been shown in recent studies that this assumption does not hold true in practice since such patches that pass the repair test suite are actually still incorrect [15], [16]. This shows that using a repair test suite alone is a weak proxy for assessing patch correctness.

Motivated by the above serious concern, researchers have employed new methods to assess patch correctness: (1) Author annotation, in which authors of repair techniques manually check the correctness of patches generated by their and competing tools by themselves – see for example [20], [22]; (2) Automated annotation by independent test suite (ITS) generated by automatic test case generation tool – see for example [17], [19]. Both methods assume that a reference (correct) implementation of the buggy program, which is used as a basis for comparison, is available. Since most ASR techniques try to fix buggy versions of real programs, the reference implementations can be found in the version control systems of the corresponding projects.

Early work that uses annotation by automatically-generated

ITS, e.g., [17], uses general-purpose automatic test generation tools such as KLEE [33] to generate an ITS that maximizes the coverage of the reference implementation written in the C programming language. Test cases generated on the reference (correct) implementation are then used to assess correctness of machine-generated patches, i.e., a machine-generated patch is regarded as incorrect if there exists a test case exposing behavioral differences in correct and machine-patched code.

Recently, Xin et al. proposed DIFFTGEN, a test generation tool for Java programs specifically designed to generate tests that can identify incorrect patches generated by ASR tools [25]. DIFFTGEN attempts to generate test cases that cover the syntactic and semantic differences between the machine-patched and human-patched programs. If there are any such test cases that expose the differences in outputs of the programs, the machine-generated patch is deemed as *incorrect* since it results in a different output as compared to the corresponding ground truth human-patched program. DIFFTGEN has been shown to be able to identify incorrect patches produced by various state-of-the-art ASR tools such as GenProg [9], Kali [15], Nopol [8], and HDRepair [34].

Best practices in building gold sets: To build gold sets objectively, a common approach is to employ many independent annotators and measure inter-rater agreement as proxy for annotation quality [27], [35]. The information retrieval (IR) community, especially through the Text REtrieval Conference (TREC)[1], has employed many annotators through a large scale collaborative effort to annotate many document corpora for various retrieval tasks. Many past software engineering studies have also involved independent annotators to construct gold sets. Based on the nature of various tasks, annotators include non-authors who could be undergraduate/graduate students [36]–[40] or professional developers [36], [41], [42].

III. USER STUDY

We conducted a user study with 35 professional developers to collect correctness labels of patches. In this study, every developer is required to complete several tasks by judging whether patches generated by ASR tools are semantically equivalent to ground truth human patches.

Patch Dataset. Since the eventual goal of our study is to assess the reliability of author and automated annotations, we need a set of patches that have been labeled before by ASR tool authors and can be used as input to automated test case generation tools designed for program repair. We find the sets of patches recently released by Xiong et al. [21], Martinez et al. [24], and Le et al. [22] to be suitable. Xiong et al. and Martinez et al. labelled a set of 210 patches generated by ASR tools designed by their research groups (i.e., ACS [20], and Nopol [8]) and their competitors (i.e., GenProg [9], Kali [15]). Le et al. labelled a set of 79 patches generated by their ASR tool (i.e., S3 [22]) and its competitors (i.e., Angelix [4], and Enumerative and CVC4 embedded in JFix [13]). The authors labelled these patches by manually comparing them with ground truth patches obtained from version control systems of the corresponding buggy subject programs. These patches can be used as input to DIFFTGEN, which is a state-of-the-art test generation tool specifically designed to evaluate patch correctness [25], and RANDOOP – a popular general purpose test case generation tool [26].

[1]http://trec.nist.gov/

TABLE I
SELECTED PATCHES AND THEIR AUTHOR LABELS

	GenProg	Kali	Nopol	ACS	S3	Angelix	Enum	CVC4
Incorrect	14	14	84	4	0	7	6	6
Correct	4	1	6	14	10	2	4	4
Unknown	2	2	5	0	0	0	0	0
Total	20	17	95	18	10	9	10	10

Due to resource constraints – only 35 professional developers agreed to spend an hour of their time in this user study – we cut down the dataset to 189 patches by randomly selecting these patches from the original datasets. Details of the dataset of 189 patches are shown in Table I.

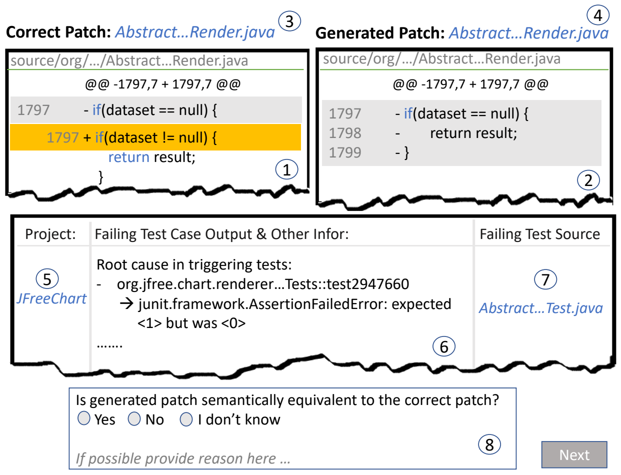
Task Design. At the start of the experiment, every participant was required to read a tutorial that briefly explains automated program repair and what they need to do to complete the tasks. Afterwards, they can complete the tasks one-by-one through a web interface.

Figure 1 shows a sample task that we give to our user study participants via our web interface. For each task, we provide a ground truth patch taken from the version control system of the corresponding buggy subject program, along with a patch that is generated by an automated program repair tool. We also provide additional resources including full source code files that are repaired by the patch, link to the GITHUB repository of the project, outputs when executing failing test cases, and source code of the failing test cases. Based on this information, participants are asked to evaluate the correctness of the patch by answering the question: *Is the generated patch semantically equivalent to the correct patch?* To answer this question, participants can choose one of the following options: "Yes", "No" or "I don't know". Finally, if they wish to, they can provide some reasons that explain their decision. Our web interface will record participants' answers and the amount of time they need to complete each task.

Participants and Task Assignment. To recruit participants, we sent emails to our industrial contacts about this user study. Our contacts then advertised the study and provided us emails of 35 developers who are willing to participate. Thirty three of the 35 professional developers participating in this study work for two large software development companies (named Company C1 and C2), while another two work as engineers for an educational institution. Company C1 currently has more than 500 employees and Company C2 has more than 2000 employees. Both companies have a large number of active projects that expose developers to various business knowledge and software engineering techniques. All the 35 developers work for projects that use Java as the main programming language.

The average number of years of work experience that these participants have is 3.5. The two developers from the educational institution are senior and have worked for 5.5 and 10 years, respectively. The most experienced developer from industry has worked for seven years, while some has only worked for one year. Participants are classified into two groups, *junior* and *senior*, according to their years of experience following the company's internal classification. Companies that our participants work for consider developers with less than 3 years of experience as juniors and those with more than 3 years of experience as seniors. There are 20 *junior* developers and 15 *senior* developers.

We divided the 35 participants into seven groups. The ratio of *junior* and *senior* developers for each group was kept approximately the same. Each patch generated by program repair tools is labeled by five participants. Participants in the same group receive the same set of patches to label.



Correct Patch: Abstract...Render.java
3
Generated Patch: Abstract...Render.java
4
1
2

```
source/org/.../Abstract...Render.java
@@ -1797,7 + 1797,7 @@
- if(dataset == null) {
+ if(dataset != null) {
return result;
}
```

```
source/org/.../Abstract...Render.java
@@ -1797,7 + 1797,7 @@
- if(dataset == null) {
-     return result;
- }
```

Project:
Failing Test Case Output & Other Infor:
Failing Test Source
5
JFreeChart
6
7
Abstract...Test.java

```
Root cause in triggering tests:
- org.jfree.chart.renderer...Tests::test2947660
  → junit.framework.AssertionFailedError: expected
    <1> but was <0>
.......
```

Is generated patch semantically equivalent to the correct patch?
○ Yes ○ No ○ I don't know

If possible provide reason here ...

Fig. 1. A sample task on our web interface. (1) and (2) show developer- and machine-generated patches; (3) and (4) show links to patched source files; (5) shows GitHub repository; (6) and (7) show output of failed test cases and their source files; (8) is the question we asked a participant.

IV. Assessing Independent Annotators' Labels

The user study presented in Section III was conducted to build a set of gold standard labels for machine-generated patches, which can reliably be used to assess reliability of author and automated annotations. Before using the labels produced by our user study, we need to first ascertain their quality. Agreement among annotators is often used as a measure of quality [27], [28], [43]. Thus, in this section, we investigate the degree to which the annotators agree with one another. This answers *RQ1: Can independent annotators agree on patch correctness?*

Methodology. To answer RQ1, we first compute some simple statistics highlighting the number of agreements and disagreements among annotators. We then calculate several well-accepted measures of inter-rater reliability. Finally, we perform some sanity checks to substantiate whether or not annotators are arbitrary in making their decisions.

TABLE II
Results of Participant Annotations

	All Agree	All Agree - Unk	Majority Agree
Incorrect	95	132	152
Correct	23	23	35
Total	118	155	187

Results. To recap, our annotators are 35 professional developers who are tasked to annotate 189 machine-generated patches. Each patch is annotated by five professional developers; each provides either one of the following labels: incorrect, correct, or unknown. Table II summarizes the number of agreements and disagreements among annotators. In the first column (All Agree), the number of patches in which all developers agree on each patch's label is 118 (62.4% of all patches); of which 95 patches are labeled as incorrect and 23 patches are labeled as correct. In the second column (All Agree - Unk), ignoring unknown labels, the number of patches for which the remaining annotators fully agree on their labels is 155 (82.0% of all patches). Out of these, the numbers of patches that are labeled as incorrect and correct are 132 and 23, respectively. In the last column (Majority Agree), for 187 out of 189 patches (98.9% of all patches), there is a majority decision (i.e., most annotators agree on one label). Out of these, 152 and 35 patches are identified as incorrect and correct, respectively.

We also compute several inter-rater reliability scores: mean pairwise Cohen's kappa [27], [44] and Krippendorff's alpha [45]. Using the earlier test we consider three different ratings (i.e., correct, incorrect, and unknown), while the latter test, which allows different number of ratings for each data point, enables us to ignore unknown ratings. Inter-rater reliability scores measure how much homogeneity, or consensus, there is between raters / labelers. The importance of rater reliability hinges on the fact that it represents the extent to which the data collected in the study are correct representations of the variables being measured. A low inter-rater reliability suggests that either the rating scale used in the study is defective or raters need to be retrained for the rating task or the task is highly subjective. The higher the inter-rater reliability the more reliable the data is.

Reliability score values by Landis and Koch [46] suggest that moderate, substantial, and almost perfect agreements are associated with values in ranges [0.41,0.60], [0.61,0.80], and [0.81,1.00] respectively. Scores below 0.41 indicate fair, slight, or poor agreements. It is worth noting that there is another interpretation of kappa value by Manning *et al.* [27], which indicates that a kappa value falling between 0.67 and 0.8 demonstrates a fair agreement between raters – the second highest level of agreement by their interpretation. It has been shown that this fair level of inter-rater agreement normally happens in popular datasets such as those used for: (1) evaluations on Text REtrieval Conference (TREC), which is championed by US National Institute of Standards and Technology (NIST) since 1992 and provides benchmark datasets for various text retrieval tasks – see http://trec.nist.gov/data.html, and (2) medical information retrieval collections [27]. Based on this interpretation,

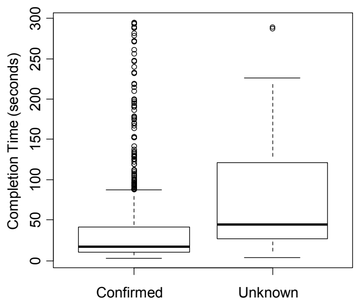


Fig. 2. Time taken by annotators to decide whether a patch's label is either known (confirmed as correct or incorrect) or unknown.

we have the following findings on the gold set annotated by independent developers:

> The computed mean pairwise Cohen's kappa and Krippendorff's alpha for our independent annotators' labels are 0.691 and 0.734 respectively. These scores indicate a substantial agreement among participants, which satisfies the standard normally met by quality benchmark datasets.

We further perform two sanity checks to substantiate whether or not annotators are arbitrary in their decisions. First, we expect conscientious annotators to spend more time inspecting patches that are eventually labeled as unknown than other patches. Annotators who label patches as unknown without thinking much would be likely making arbitrary decisions. Figure 2 depicts a box plot showing the time participants took on patches that are labeled as known (correct or incorrect) or unknown. It can be seen that participants took more time on the later set of patches. Wilcoxon signed-rank test returns a p-value that is less than 0.005, indicating a statistically significant difference. Moreover, the Cliff's delta, which is a non-parametric effect size measure, is 0.469 (medium).

Second, we expect conscientious annotators to spend more time inspecting difficult patches than easy ones. We consider disagreement among annotators as a proxy for patch difficulty. We compare the time taken by participants in identifying patches for which there is complete agreement to those for which disagreement exists. Figure 3 shows a box plot which shows that participants spend more time on disagreement cases. Wilcoxon signed-rank test returns a p-value that is less than 0.05, indicating statistically significant difference. Moreover, the Cliff's delta is 0.178 (small).

The above results substantiate the quality of our dataset. In the subsequent sections, which answer RQ2 and RQ3, we use two versions of our dataset, ALL-AGREE (see "All Agree" column in Table II) and MAJORITY-AGREE (see "Majority Agree" column in Table II), to assess the reliability of author and automated annotations.

V. Assessing Author Annotation

A number of studies proposing automated repair approaches evaluate them through manual annotation performed by authors, e.g, [20], [34]. Author subjectivity may cause bias which can be a threat to the internal validity of the study. Author bias has been actively discussed especially in the medical domain, e.g., [48]. Unfortunately so far, there has been no study that investigates presence or absence of bias in author annotation and its impact to the validity of the labels in automated repair. This section describes our effort to fill this need by answering *RQ2: How reliable is author annotation?*

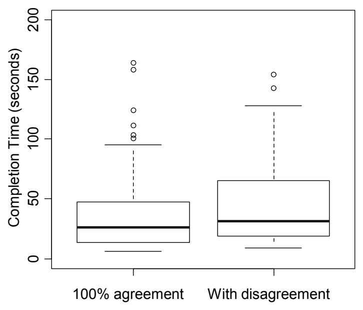


Fig. 3. Time taken by annotators to decide a patch's label for full-agreement and disagreement cases.

TABLE III
Independent (Indep) Annotator vs. Author Labels

	Indep Annotators-Authors	ALL-AGREE	MAJORITY-AGREE
Same	Incorrect-Incorrect	82	133
	Correct-Correct	23	33
Different	Incorrect-Correct	6	10
	Correct-Incorrect	0	2
	Incorrect-Unknown	7	9
	Correct-Unknown	0	0
Total		118	187

Methodology. Recall that our user study makes use of patches released by three research groups, including Xiong et al. [21], Martinez et al. [24], and Le et al. [22] who created program repair tools namely ACS, Nopol, and S3 respectively. Authors of each tool manually labeled the patches generated by their tool and competing approaches by themselves. To answer RQ2, we compare labels produced by the three research groups with those produced by our independent annotators whose quality we have validated in Section IV. We consider the ALL-AGREE and MAJORITY-AGREE datasets mentioned in Section IV.

Results. Table III shows the detailed results on the comparisons between independent annotators' and authors' labels. We found that for ALL-AGREE dataset, authors' labels match with independent annotators' labels (**Same**) for 105 out of 118 patches (89.0%). There are 13 patches for which authors' labels mismatch those by independent annotators (**Different**). Among these patches, 6 are identified by independent annotators as incorrect, but identified by authors as correct (Incorrect-Correct). For the other 7 patches, authors' labels are unknown while independent annotators' labels are incorrect (Incorrect-Unknown). For the MAJORITY-AGREE dataset, 88.8% of the labels match. There are 21 mismatches; 10 belong to Incorrect-Correct cases, 2 to Correct-Incorrect cases, and 9 to Incorrect-Unknown cases. Figure 4 shows an example

```
 1  @@ -115,9 +115,7 @@ public class StopWatch {
 2  public void stop() {
 3     if(this.runningState != STATE_RUNNING && this.
          runningState != STATE_SUSPENDED) {
 4        throw new IllegalStateException("...");
 5     }
 6  +  if(this.runningState == STATE_RUNNING)// Developer
        patch
 7  +  if(-1 == stopTime)// Generated patch
 8        stopTime = System.currentTimeMillis();
 9     this.runningState = STATE_STOPPED;
10  }
```

Fig. 4. An example of a patch that has mismatched labels. Xiong et al. identified the patch (shown at line 7) as correct, while independent annotators identified this patch as incorrect. The ground truth (developer) patch is shown at line 6.

patch generated by Nopol [8] that has mismatched labels. It is labeled as correct by Martinez et al. and incorrect by independent annotators.

We also compute inter-rater reliability of authors' labels and labels in ALL-AGREE and MAJORITY-AGREE datasets. The Cohen's kappa values are 0.719 and 0.697 considering the ALL-AGREE and MAJORITY-AGREE datasets respectively. The Krippendorf's alpha values are 0.717 and 0.695. Comparing these scores with Landis and Koch's interpretation described in Section IV, there is substantial agreement.

A majority (88.8-89.0%) of patch correctness labels produced by author annotation match those produced by independent annotators. Inter-rater reliability scores indicate a substantial agreement between author and independent annotator labels.

To characterize cases where author and independent annotator labels match (**Same**) and those where they do not match (**Different**), we investigate the time that participants of our user study took to label the two sets of patches. Since the number of mismatches is smaller in the ALL-AGREE dataset, we focus on comparing labels in MAJORITY-AGREE dataset. Figure 5 depicts a box plot showing the distribution of completion time corresponding to the two sets of patches. The figure shows that patches with matching labels took participants a shorter period of time to label comparing to those whose labels mismatched. Wilcoxon signed-rank test returns a p-value that is less than 0.05, indicating statistically significant difference. The Cliff's delta is equal to 0.278 (small). Since task completion time can be used as a proxy for measuring task difficulty or lack thereof [49], we consider participants completion time as a proxy of difficulty in assessing patch correctness. The result suggests that disagreements between authors and independent annotators happen for difficult cases.

VI. Assessing Automated Annotation

We also investigate the reliability of the use of automatically generated independent test suite (ITS) in annotating patch labels. ITS has been used as an objective proxy to measure patch correctness – a patch is deemed as incorrect if it does not pass the ITS, and as correct or generalizable otherwise [17], [19]. It is unequivocal that incorrect patches determined by ITS

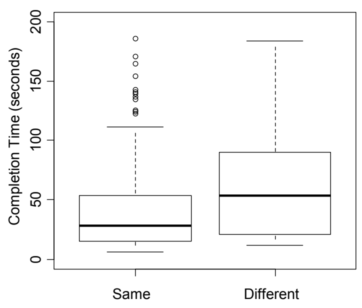


Fig. 5. Participant completion time for patches for which author and independent annotator labels match (**Same**) and mismatch (**Different**)

are indeed incorrect. However, it is unclear if ITS can detect a large proportion of incorrect patches. Moreover, the extent to whether correct (generalizable) patches determined by ITS are indeed correct remains questionable. Thus, to assess the usefulness of ITS, we investigate the answer to *RQ3: How reliable is automatically generated ITS in determining patch correctness?*

Methodology: We employ the recently proposed test case generation tool DiffTGen by Xin et al. [25] and Randoop [26] to generate ITS. To generate ITS using DiffTGen and Randoop, the human-patched program is used as ground truth. For DiffTGen, we run using its best configuration reported in [25], allowing it to invoke EvoSuite [50] in 30 trials with the search time of each trial limited to 60 seconds. A machine-generated patch is identified as *incorrect* if there is a test in the DiffTGen-generated ITS that witnesses the output differences between the machine and human patches. For Randoop, we run it on the ground truth program with 30 different seeds with each run limited to 5 minutes. A machine-generated patch is identified as incorrect if there is at least one test case in the Randoop-generated ITS that exhibits different test results in machine-patched and human-patched (ground truth) programs, e.g., it fails on the machine-patched program but passes on the ground truth program, or vice versa. By this way, we allow both tools to generate multiple test suites. It is, however, worth noting that DiffTGen and Randoop are incomplete in the sense that they do not guarantee to always generate the test cases that witness incorrect patches.

We use test cases generated by the tools to automatically annotate the 189 patches and compare the generated labels to those in ALL-AGREE and MAJORITY-AGREE datasets which are created by our user study.

Results: Out of the 189 patches in our study, DiffTGen generates test cases that witness 27 incorrect (overfitting) patches. Details of these patches are shown in Table V. The ALL-AGREE ground truth identifies 17 of these 27 patches as incorrect (the other 10 patches lie outside of the ALL-AGREE dataset), while the MAJORITY-AGREE dataset identifies all of them as incorrect. Unfortunately, most of the patches labelled as incorrect in ALL-AGREE (65 patches) and MAJORITY-AGREE (121 patches) datasets failed to be detected as such by ITS generated by DiffTGen. Randoop performs similarly as compared to DiffTGen. It identifies

TABLE IV
KAPPA AND ALPHA VALUES WHEN USING DIFFTGEN, RANDOOP, AND THEIR COMBINATION TO LABEL PATCHES

	ALL-AGREE			MAJORITY-AGREE		
	DIFFT	RAND	COMB	DIFFT	RAND	COMB
Cohen's Kappa	0.078	0.073	0.158	0.075	0.072	0.146
Kripp's Alpha	-0.32	-0.3	-0.057	-0.336	-0.313	-0.097

31 patches as incorrect, all of which are also identified as incorrect in the MAJORITY-AGREE dataset. Note that, DIFFTGEN and RANDOOP when combined can identify totally 51 unique patches as incorrect. For each of the total 189 patches, DIFFTGEN and RANDOOP generated from 1186 to 3619 unit test cases per method. There are a few patches that the tools cannot generate test cases for.

In their studies, Smith et al. [17] and Le et al. [17] assume a patch is incorrect if it does not pass an ITS, and correct or generalizable otherwise. Using the same assumption to generate correctness labels, we can compute inter-rater reliability between labels automatically annotated by running ITS generated by DIFFTGEN and RANDOOP and labels in ALL-AGREE and MAJORITY-AGREE datasets. As readers may have expected, the Cohen's kappa values are very low as shown in Table IV, e.g., kappa values when using DIFFTGEN-generated ITS for ALL-AGREE and MAJORITY-AGREE are 0.078 and 0.075 respectively. The corresponding Krippendorff's alpha values are -0.32 and -0.336.

We now compare author labels discussed in Section V with ITS labels. Table V shows the author labels of the 27 and 31 patches identified as incorrect by DIFFTGEN and RANDOOP, respectively. For these patches, the majority of the labels by authors and DIFFTGEN match. However, interestingly, there are four special patches in which labels generated by automated- and author-annotations are mismatched. These cases are highlighted in gray in Table V. Particularly, three patches are identified as incorrect by DIFFTGEN, including Math_80 generated by Kali, Chart_3 generated by GenProg, and Math_80_2015 generated by Nopol, while author labels are "Unknown". One patch identified as incorrect by RANDOOP (Math_73 generated by GenProg), is labelled as correct by authors. Based on results above, we conclude:

> Independent test suites generated by DIFFTGEN and RANDOOP can only label fewer than a fifth of incorrect patches as such in ALL-AGREE and MAJORITY-AGREE datasets. However, generated test suites can be used as a complement for author annotation to increase accuracy.

Finally, we want to investigate the difficulty of judging correctness of patches that are labelled as incorrect by ITSs generated by DIFFTGEN and RANDOOP. To do so, we compare participant completion time for the set of 51 unique patches and another set containing the other patches. We find that they are more or less the same. Wilcoxon signed-rank test confirms that the difference is not statistically significant. Thus, patches that ITS successfully labels as incorrect are not necessarily the ones that participants require more time to manually label.

TABLE V
LABELS BY INDEPENDENT ANNOTATORS ("ANNOT" COLUMN) AND AUTHORS ("AUTHORS" COLUMN) OF PATCHES IDENTIFIED BY INDEPENDENT TEST SUITE (ITS) GENERATED BY DIFFTGEN OR RANDOOP AS INCORRECT .

		DIFFTGEN	**RANDOOP**	**Annot**	**Authors**
Kali	Time_4	Incorrect	Incorrect	Incorrect	Incorrect
	Math_32	Incorrect		Incorrect	Incorrect
	Math_2	Incorrect		Incorrect	Incorrect
	Math_80	Incorrect		Incorrect	Unknown
	Math_95	Incorrect	Incorrect	Incorrect	Incorrect
	Math_40	Incorrect		Incorrect	Incorrect
	Chart_13	Incorrect		Incorrect	Incorrect
	Chart_26	Incorrect		Incorrect	Incorrect
	Chart_15	Incorrect	Incorrect	Incorrect	Incorrect
	Chart_5	Incorrect	Incorrect	Incorrect	Incorrect
GenProg	Math_2	Incorrect		Incorrect	Incorrect
	Math_8	Incorrect		Incorrect	Incorrect
	Math_80	Incorrect		Incorrect	Incorrect
	Math_81	Incorrect		Incorrect	Incorrect
	Math_95	Incorrect	Incorrect	Incorrect	Incorrect
	Math_40	Incorrect		Incorrect	Incorrect
	Math_73		Incorrect	Incorrect	Correct
	Chart_1		Incorrect	Incorrect	Incorrect
	Chart_3	Incorrect		Incorrect	Unknown
	Chart_5	Incorrect	Incorrect	Incorrect	Incorrect
	Chart_15	Incorrect	Incorrect	Incorrect	Incorrect
Nopol	Math_33	Incorrect		Incorrect	Incorrect
	Math_73_2017		Incorrect	Incorrect	Incorrect
	Math_80_2017	Incorrect		Incorrect	Incorrect
	Math_80_2015	Incorrect		Incorrect	Unknown
	Math_97	Incorrect		Incorrect	Incorrect
	Math_105		Incorrect	Incorrect	Incorrect
	Time_16		Incorrect	Incorrect	Incorrect
	Time_18		Incorrect	Incorrect	Incorrect
	Chart_13_2017	Incorrect		Incorrect	Incorrect
	Chart_13_2015	Incorrect		Incorrect	Incorrect
	Chart_21_2017	Incorrect		Incorrect	Incorrect
	Chart_21_2015	Incorrect		Incorrect	Incorrect
	Closure_7		Incorrect	Incorrect	Incorrect
	Closure_12		Incorrect	Incorrect	Incorrect
	Closure_14		Incorrect	Incorrect	Incorrect
	Closure_20		Incorrect	Incorrect	Incorrect
	Closure_30		Incorrect	Incorrect	Incorrect
	Closure_33		Incorrect	Incorrect	Incorrect
	Closure_76		Incorrect	Incorrect	Incorrect
	Closure_111		Incorrect	Incorrect	Incorrect
	Closure_115		Incorrect	Incorrect	Incorrect
	Closure_116		Incorrect	Incorrect	Incorrect
	Closure_120		Incorrect	Incorrect	Incorrect
	Closure_124		Incorrect	Incorrect	Incorrect
	Closure_130		Incorrect	Incorrect	Incorrect
	Closure_121		Incorrect	Incorrect	Incorrect
	Mockito_38		Incorrect	Incorrect	Incorrect
Angelix	Lang_30		Incorrect	Incorrect	Incorrect
CVC4	Lang_30		Incorrect	Incorrect	Incorrect
Enum	Lang_30		Incorrect	Incorrect	Incorrect

VII. DISCUSSION

In this section, we first provide implications of our findings. We then discuss our post-study survey, in which we asked a number of independent annotators for rationales behind their patch correctness judgements. Future work and possible challenges inspired by our study are described next. At the end of this section, we discuss some threats to validity.

A. Implications

To recap, we have gained insights into the reliability of patch correctness assessment by authors and by automatically generated independent test suite (ITS); each of them has their own advantages and disadvantages. Based on these insights, we provide several implications as follows:

> Authors' evaluation of patch correctness should be made publicly available to the community.

Xiong et al., Martinez et al., and Le et al. released their patch correctness labels publicly [21], [22], [24], which we are grateful for. We believe that considerable effort has been made by authors to ensure the quality of the labels. Still, we noticed that for slightly more than 10% of the patches, authors' labels are different from the ones produced by multiple independent annotators. Thus, we encourage future ASR paper authors to release their datasets for public inspection. The public (including independent annotators) can then provide inputs on the labels and possibly update labels that may have been incorrectly assigned. Our findings here (e.g., author annotations are fairly reliable) may not generalize to patches labelled by authors which have not been released publicly. It is possible that the quality of correctness labels for those patches (which are not made publicly available) to be lower. Also, as criticized by Monperrus *et al.* [51], the conclusiveness of the evaluation of techniques that keep patches and their correctness labels private is questionable.

> Collaborative effort is needed to distribute the expensive cost of ASR evaluation.

In this study, we have evaluated correctness of 189 automatically generated patches by involving independent annotators. We have shown that the quality of the resultant labels (measured using inter-rater reliability) are on par with high-quality text retrieval benchmarks [27]. Unfortunately, evaluation using independent annotators is expensive. To evaluate 189 patches, we needed to get 35 professional developers; each agreed to spend up to an hour of their time. This process may not be scalable especially considering the large number of new ASR techniques that are released in the literature year by year. Thus, there is a need for more collaborative effort to distribute the cost of ASR evaluation. One possibility is to organize a competition involving impartial industrial data owners (e.g., software development houses willing to share some of their closed bugs) who are willing to judge correctness of generated patches. Similar competitions with industrial data owners have been held to advance various fields such as forecasting[2] and fraud detection[3].

> Independent test suite (ITS) *alone* should not be used to evaluate the effectiveness of ASR.

Independent test suites (ITSs) generated by DiffTGen [25] and Randoop [26] have been shown to be ineffective in annotating correctness labels for patches (see Section VI). Only fewer than a fifth of the incorrect patches are identified as such by ITSs generated by DiffTGen and Randoop. Based on effectiveness of state-of-the-art test generation tool for automatic repair that we assessed in this study, we believe that ITS *alone* should not be used for *fully automated* patch labeling. The subject of ITS generation for program repair is new though and we encourage future studies to improve the quality of automatic test generation tools so that more incorrect

[2]http://www.cikm2017.org/CIKM_AnalytiCup_task1.html
[3]http://research.larc.smu.edu.sg/fdma2012/

```
@@ -168,7 +168,7 @@ public class PearsonsCorrelation {
         } else {
             double r = correlationMatrix.getEntry(i, j);
             double t = Math.abs(r * Math.sqrt((nObs - 2)/(1 - r * r)));
+            out[i][j] = 2 * tDistribution.cumulativeProbability(-t);
-            out[i][j] = 2 * (1 - tDistribution.cumulativeProbability(t));
         }
```

(a) Human Patch

```
@@ -190,6 +190,7 @@
         for (int j = 0; j < i; j++) {
           double corr = correlation(matrix.getColumn(i), matrix.getColumn(j));
           outMatrix.setEntry(i, j, corr);
+          if(1 - nVars < -1)
           outMatrix.setEntry(j, i, corr);
         }
```

(b) Generated Patch

Fig. 6. A machine-generated patch labeled by ITS as incorrect but labeled by author annotation as unknown.

patches can be detected. That being said, automated patch annotation may not be a silver bullet; the general problem of patch correctness assessment (judging the equivalence of developer patch and automatically generated patch) is a variant of program equivalence problem which has been proven to be undecidable with no algorithmic solution [52].

> Independent test suite, despite being less effective, can be used to augment author annotation.

It has been shown in Section VI that ITS generated by DiffTGen and Randoop identified four patches as incorrect whereas the labels generated by author annotation were unknown or correct. An example of such a patch is shown in Figure 6. From the figure, we can notice that it is hard to manually determine whether the patch is correct or not. From this finding, we believe that ITS, despite being less effective than author annotation in identifying correct patches, can be used to augment author annotation by helping to resolve at least some of the ambiguous cases. Authors can possibly run DiffTGen and Randoop to identify clear cases of incorrect patches; the remaining cases can then be manually judged. The use of both author and automated annotation via ITS generation can more closely approximate multiple independent annotators' labels while requiring less cost.

B. Post-Study Survey

We conducted a post-study survey to investigate why a developer chooses a different answer from the majority. Among the 189 patches, there are several patches where the majority, but not all participants, agree on patch correctness. Among participants annotating these patches, we selected 11 who answered differently from the majority and emailed them to get deeper insights into their judgments. In our email, we provided a link to the same web interface used in our user study to allow participants to revisit their decision for the patch in question. Notice that we did not inform the participants that their answers were different from the majority. We received replies from 8 out of the 11 participants (72.7% response rate).

We found that 5 out of 8 developers changed their correctness labels after they looked into the patch again; their revised labels thus became consistent with the labels that the majority agree. The remaining three kept their correctness labels; two judged two different patches as incorrect (while the majority labels are correct) while another judged a patch as correct (while the majority label is incorrect). These participants kept their decision for different reasons; one was unsure of a complex expression involved in the patch, another highlighted a minor difference that may be considered ignorable by others, and the other participant viewed the generated and ground truth patch to have similar intentions.

C. Future Extensions

Beyond program repair. The contribution of this work is an empirical investigation on the reliability of popular evaluation methods followed in past studies on program repair.

We believe that this kind of meta-study that assesses reliability of evaluation methods should also be performed beyond program repair, in areas such as software mining, fault localization, defect prediction, static analysis, and others, that require a validation of results. Often past studies involve performance assessment made by authors done by, e.g., manually or semi-automatically labelling the results [53]–[55] or based on historical data that are dirty [56], [57]. Effort should be made for a more rigorous assessment (which may be more costly) to see if biases exists (with the cheaper and existing evaluation alternatives) and if biases exist, the extent to which they exist. We believe that our work can provide valuable insights in the design of these future studies.

There have been already efforts done in this area – studies that investigate bias in software engineering [56]–[59]. Our work is unique compared to these existing studies in terms of the target task investigated (i.e., ASR) and the methodology employed (e.g., the use of multiple independent professionals as annotators). These studies are a good start but much more work is needed to ensure that current assessment methods employed to evaluate performance of many existing research solutions correctly reflect the quality of underlying tools being assessed.

Usage of specifications. In this work, we used labels by independent annotators as ground truth to assess reliability of author- and automated-annotations. Independent annotators are, however, still humans and can admittedly make mistakes even with a substantial amount of time devoted to the annotation task. To avoid this threat, complete and correct specifications can be used in conjunction with a sound static verifier to serve as a reliable patch validation method, e.g., a patch passing the verification is definitely a correct one [60]. This could be achieved by creating a benchmark of programs equipped with complete and correct specifications and a set of test cases. Test cases can then be used by program repair techniques to generate patches and those machine-generated patches can then be validated against specifications using a sound verifier. We plan to investigate this direction by using the OpenJML verifier [61] on programs accompanied by JML annotations [62]. Although complete and correct specifications are hard to obtain in practice, a study with such specifications would be worth exploring since by doing so the extent to which a program repair technique overfits to test suite used for repair can be unequivocally determined. To make this possible, we plan to tradeoff the scale of studied systems for a higher degree of soundness in patch assessment.

D. Threats to Validity

Threats to internal validity. These threats relate to potential errors and biases in our study. We discuss them below:

To reduce the threat of potential errors in our code, we conducted a pilot study with a few graduate students and thoroughly checked our code.

We do not use all patches in the original dataset by Xiong et al. [21], Martinez et al. [24], and Le et al. [22] due to constrained resources (we only have 35 professional developers agreeing to devote an hour of their time; the number is similar to those of past studies [59], [59]). The results may differ if the whole dataset is used. To mitigate this threat, we randomly selected patches included in this study while keeping the ratios of patches generated by ASR tools approximately the same.

The professional developers that we employed are not the original developers of the buggy code and ground truth patches. Unfortunately, since the original developer patches included in Xiong et al.'s study were committed many years ago (the earliest being 2006), it is hard to contact those developers. Even if we can involve them, they may have forgotten the detail of the patches. However, since the patches are small, professional developers participated in our study should be able to assess patch correctness. Indeed, in our study, respondents were able to provide definite labels to a majority of patches (i.e., only 5.9% are unknown, while the rest are either incorrect or correct). Additionally, we asked not only one professional developer but five of them to label each patch. Section IV highlights that there is a substantial agreement among participants, which is on par with high-quality benchmark datasets. Moreover, participants are provided with multiple resources, e.g., source code files, failed test cases, GITHUB link of the project, etc, for the annotation task. A large number of past software engineering studies e.g., [37], [38], [41], [63]–[65] has also involved third-party labelers (who are not content creators) to assign labels for data. And the same annotation setup was also followed in other related areas, e.g., information retrieval [28], [66]. Last but not least, we also make the 189 patches and participants' responses publicly available for public inspection [67].

Threats to external validity. These threats relate to the generalizability of our results. We discuss them below:

We included 189 patches generated by 8 ASR tools to fix buggy code from 13 software projects. We believe this is a substantial number of patches generated by a substantial number of state-of-the-art ASR tools. Past empirical studies on ASR, e.g., [15], include five tools and 55 patches from 105 bugs. Still, we acknowledge that results may differ if more patches, projects and ASR tools are considered.

We have included 35 professional developers in our user study. This number is larger or similar to those considered in many prior work, e.g., [68]–[70]. The results may differ for other groups of developers. To reduce this threat, we have selected a mix of junior and senior developers from two large IT companies and a large educational institution.

Threats to construct validity. These threats relate to the suitability of our evaluation metrics. In this study, we use average pairwise Cohen's kappa and Krippendorff's alpha to evaluate the reliability of the patch labels from independent annotators. We also use the two to measure agreement between independent annotators' labels and those produced by author and automated annotations. These metrics are widely used in many research areas, e.g., information retrieval [71]–[73], software engineering [74], [75], etc. Thus, we believe there is little threat to construct validity.

VIII. Related Work

Program repair. There are several ASR techniques beyond those investigated in our study: RSRepair [76] and AE [31] are random search techniques. PAR [10] uses templates to repair. Prophet [7] and HDRepair [34] use historical bug fix data to guide the repair process. SemFix [77], DirectFix [3], and SPR [6] use symbolic execution and angelix debugging. Qlose [78] use program traces to rank repairs in the order of likelihood of being correct. Elixir [79] uses machine learning to generate repairs. Jaid [80] builds rich abstraction state for repair. We refer interested readers to Gazzola et al.'s survey paper [81] for a more comprehensive review.

Patch correctness assessment. Qi et al. [15] empirically studied patches generated by GenProg [9], RSRepair [32], and AE [31]. They manually investigated the patches, wrote additional test cases, and reported the results on running the patches against additional test cases. Authors of PAR [10] performed a user study on the acceptability of patches generated by their tool. They employed 89 students and 164 developers to confirm that patches generated by PAR are more acceptable than GenProg. Monperrus *et al.* [51] discuss the main evaluation criteria of automatic software repair including understandability, correctness and completeness. They suggest that repair techniques having their generated patches along with correctness labels kept private, such as PAR, are questionable. To avoid potential bias of manual human investigation, Smith *et al.* use automatic test case generation tool KLEE [82] to generate independent test suites (ITS) that maximize coverage of ground-truth program to assess machine-generated patches [17]. Using ITS, they evaluate the effectiveness of GenProg, RSRepair (aka. TrpAutoRepair), and AE on the IntroClass dataset [83]. Recently, Xin et al. [25] and Xiong et al. [21] proposed an automated approach to identify incorrect machine-generated patches via execution traces. They leverage automatic test generation to generate additional test cases, and use execution traces when executing test cases to determine whether a machine-generated patch is correct or incorrect.

Unlike previous works which compare and evaluate effectiveness of ASR solutions, the main goal of our study is to assess whether methodologies that are often used for effectiveness evaluation of ASR are fair or reliable. We do this by assessing reliability of author annotation and automated annotation by using a gold set of labels collectively built by professional developers following standard best practice. .

Empirical studies on biases and reliability. Bird *et al.* highlighted that only a fraction of bug fixes are labelled in version control systems and this causes a systematic bias in the evaluation of defect prediction tools [56]. Herzig *et al.* manually examined 7,000 reports from issue tracking systems of open source projects and reported that 33.8% of all bug reports to be misclassified [84]. They showed that the misclassification introduces bias to defect prediction studies since a substantial number of files is wrongly marked as defective. The goal of our study is similar to the goals above – we want to highlight and reduce bias in the evaluation of automated software engineering tools.

IX. Conclusion and Future Work

We assessed the reliability of existing patch correctness assessment methods via a user study. The study involved 35 professional developers and resulted in a high-quality gold set of correctness labels for 189 patches generated by different ASR techniques. Using the gold set, we assess reliability of author annotation (i.e., Xiong et al. [21], Martinez et al. [24], and Le et al. [22]) and automated annotation (i.e. DiffTGen [25] and Randoop [26]). We find that: (1) A majority (88.8-89.0%) of labels produced by authors match those produced by independent annotators, (2) Only fewer than a fifth of incorrect patches can be labelled by DiffTGen and Randoop as such. DiffTGen and Randoop can, however, uncover multiple incorrect patches labeled as "unknown" or "correct" by authors. Based on our findings, we recommend that ASR authors publicly release their labels, and that more collaborative effort to distribute the expensive cost of ASR evaluation. We also stressed that although ITS alone should not be used to fully judge patch correctness labels, it can be used in conjunction with author annotation to increase accuracy.

We plan to explore the extensions described in Section VII-C, and expand our gold set by recruiting more professional developers and collecting more ASR-generated patches. Organizing competitions with industrial data owners (e.g., with our two industrial partners whose developers have participated in this study) is also interesting to explore.

Acknowledgements

Xuan-Bach D. Le and Lingfeng Bao are joint first authors. Xin Xia is the corresponding author. Xuan-Bach D. Le and Corina Pasareanu are sponsored by DARPA under agreement number FA8750-15-2-0087. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] G. Tassey, "The economic impacts of inadequate infrastructure for software testing." *Planning Report, NIST*, 2002.

[2] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software," University of Cambridge, Judge Business School, Tech. Rep., 2013.

[3] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *International Conference on Software Engineering (ICSE)*. IEEE Press, 2015, pp. 448–458.

[4] ——, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 691–701.

[5] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *International Conference on Software Engineering (ICSE)*. IEEE Press, 2017, pp. 416–426.

[6] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 166–178.

[7] ——, "Automatic patch generation by learning correct code," in *Symposium on Principles of Programming Languages (POPL)*, 2016, pp. 298–312.

[8] J. Xuan, M. Martinez, F. Demarco, M. Clément, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *Transactions on Software Engineering*, 2016.

[9] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *International Conference on Software Engineering*, ser. ICSE'12, 2012, pp. 3–13.

[10] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 802–811.

[11] X.-B. D. Le, T.-D. B. Le, and D. Lo, "Should fixing these failures be delegated to automated program repair?" in *International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 427–437.

[12] X. B. D. Le, Q. L. Le, D. Lo, and C. Le Goues, "Enhancing automated program repair with deductive verification," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 428–432.

[13] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "Jfix: Semantics-based repair of java programs via symbolic pathfinder," in *International Symposium on Software Testing and Analysis*, ser. ISSTA'17, 2017 (to appear).

[14] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic debugging," in *International Conference on Software Engineering*, ser. ICSE'11, 2011, pp. 121–130.

[15] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 24–36.

[16] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 702–713.

[17] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 532–543.

[18] X.-B. D. Le, F. Thung, D. Lo, and C. L. Goues, "Overfitting in semantics-based automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 163–163.

[19] X.-B. D. Le, D. Lo, and C. Le Goues, "Empirical study on synthesis engines for semantics-based program repair," in *International Conference on Software Maintenance and Evolution*, ser. ICSME'16, 2016, pp. 423–427.

[20] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *International Conference on Software Engineering*. IEEE Press, 2017, pp. 416–426.

[21] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 789–799.

[22] X. B. D. Le, D. H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by example," *FSE. ACM*, 2017.

[23] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis*, ser. ISSTA '14, 2014, pp. 437–440.

[24] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9470-4

[25] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 226–236.

[26] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, 2007, pp. 75–84. [Online]. Available: https://doi.org/10.1109/ICSE.2007.37

[27] D. M. Christopher, R. Prabhakar, and S. Hinrich, "Introduction to information retrieval," *An Introduction To Information Retrieval*, vol. 151, p. 177, 2008.

[28] T. T. Damessie, T. P. Nghiem, F. Scholer, and J. S. Culpepper, "Gauging the quality of relevance assessments using inter-rater agreement," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval, Shinjuku, Tokyo, Japan, August 7-11, 2017*, 2017, pp. 1089–1092.

[29] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *International Conference on Software Engineering (ICSE)*, Cape Town, South Africa, 2010, pp. 215–224.

[30] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," *Dependable Software Systems Engineering*, 2015.

[31] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 356–366.

[32] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 254–265.

[33] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008, pp. 209–224.

[34] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016, pp. 213–224.

[35] L. Dybkjaer, H. Hemsen, and W. Minker, *Evaluation of Text and Speech Systems*, 1st ed. Springer Publishing Company, Incorporated, 2007.

[36] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: a case study of bug reports," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 505–514.

[37] D. D. Gachechiladze, F. Lanubile, N. Novielli, and A. Serebrenik, "Anger and its direction in apache jira developer comments," in *Proc. of the Int. Conf. on Software Engineering (ICSE)*, 2017.

[38] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.

[39] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Labeling source code with information retrieval methods: an empirical study," *Empirical Software Engineering*, pp. 1383–1420, 2014.

[40] Y. Zou, T. Ye, Y. Lu, J. Mylopoulos, and L. Zhang, "Learning to rank for question-oriented software text retrieval (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 1–11.

[41] O. Ormandjieva, I. Hussain, and L. Kosseim, "Toward a text classification system for the quality assessment of software requirements written in natural language," in *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*. ACM, 2007, pp. 39–45.

[42] C. Treude, M. P. Robillard, and B. Dagenais, "Extracting development tasks to navigate software documentation," *IEEE Transactions on Software Engineering*, vol. 41, no. 6, pp. 565–581, 2015.

[43] F. Scholer, A. Turpin, and M. Sanderson, "Quantifying test collection quality based on the consistency of relevance judgements," in *Proceeding of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2011, Beijing, China, July 25-29, 2011*, 2011, pp. 1063–1072.

[44] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[45] K. Krippendorff, "Estimating the reliability, systematic error, and random error of interval data," *Educational and Psychological Measurement*, vol. 30, no. 1, pp. 61–70, 1970.

[46] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.

[47] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," *Psychological Bulletin*, vol. 114, no. 3, p. 494, 1993.

[48] A. R. Vaccaro, A. Patel, and C. Fisher, "Author conflict and bias in research: Quantifying the downgrade in methodology," *Spine*, vol. 30, no. 14, 2011.

[49] C. D. Wickens, "Processing resources and attention," *Multiple-task performance*, vol. 1991, pp. 3–34, 1991.

[50] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13)*, 2011, pp. 416–419.

[51] M. Monperrus, "A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 234–242.

[52] M. Sipser, *Introduction to the Theory of Computation*, 1st ed. International Thomson Publishing, 1996.

[53] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 205–214.

[54] A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza, "Content classification of development emails," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 375–385.

[55] F. Thung, X.-B. D. Le, and D. Lo, "Active semi-supervised defect categorization," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 2015, pp. 60–70.

[56] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, 2009, pp. 121–130.

[57] P. S. Kochhar, Y. Tian, and D. Lo, "Potential biases in bug localization: Do they matter?" in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 803–814.

[58] C. Bird, "Dont embarrass yourself: Beware of bias in your data," in *Perspectives on Data Science for Software Engineering*. Elsevier, 2016, pp. 309–315.

[59] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 199–209.

[60] X.-B. D. Le, "Towards efficient and effective automatic program repair," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016, 2016, pp. 876–879.

[61] D. R. Cok, "Openjml: Jml for java 7 by extending openjdk," in *NASA Formal Methods Symposium*. Springer, 2011, pp. 472–479.

[62] G. T. Leavens, A. L. Baker, and C. Ruby, "Jml: a java modeling language," in *Formal Underpinnings of Java Workshop (at OOPSLA98)*, 1998, pp. 404–420.

[63] O. Baysal, R. Holmes, and M. W. Godfrey, "No issue left behind: Reducing information overload in issue tracking," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 666–677.

[64] A. J. Ko, B. Dosono, and N. Duriseti, "Thirty years of software problems in the news," in *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, 2014, pp. 32–39.

[65] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 107–118.

[66] P. Bailey, N. Craswell, I. Soboroff, P. Thomas, A. P. de Vries, and E. Yilmaz, "Relevance assessment: are judges exchangeable and does it matter," in *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2008, pp. 667–674.

[67] X.-B. D. Le, *Dataset*, 2009. [Online]. Available: https://github.com/anonymousICSE2019/patchcorrectness

[68] K. Kevic, B. M. Walters, T. R. Shaffer, B. Sharif, D. C. Shepherd, and T. Fritz, "Tracing software developers' eyes and interactions for change tasks," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 202–213.

[69] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 672–681.

[70] J. Rubin and M. Rinard, "The challenges of staying together while moving fast: An exploratory study," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 982–993.

[71] C. Castillo, D. Donato, L. Becchetti, P. Boldi, S. Leonardi, M. Santini, and S. Vigna, "A reference collection for web spam," in *ACM Sigir Forum*, vol. 40, no. 2. ACM, 2006, pp. 11–24.

[72] E. Meij, "Combining concepts and language models for information access," in *SIGIR Forum*, vol. 45, no. 1, 2011, p. 80.

[73] E. Amigó, J. Gonzalo, and F. Verdejo, "A general evaluation measure for document organization tasks," in *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2013, pp. 643–652.

[74] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng, "Detecting missing information in bug descriptions," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 396–407.

[75] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 385–395.

[76] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 254–265.

[77] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *International Conference on Software Engineering (ICSE)*. IEEE Press, 2013, pp. 772–781.

[78] L. D'Antoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantitative objectives," in *International Conference on Computer Aided Verification (CAV)*. Springer, 2016, pp. 383–401.

[79] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 2017, pp. 648–659.

[80] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 2017, pp. 637–647.

[81] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, 2017.

[82] B. Carterette and I. Soboroff, "The effect of assessor error on ir system evaluation," in *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2010, pp. 539–546.

[83] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *Transactions on Software Engineering (TSE)*, vol. 41, no. 12, pp. 1236–1256, Dec. 2015.

[84] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 392–401.