Human-In-The-Loop Automatic Program Repair

Charaka Geethal, Marcel Böhme, Van-Thuan Pham

Abstract—LEARN2FIX is a *human-in-the-loop interactive program repair* technique, which can be applied when no bug oracle—except the user who is reporting the bug—is available. This approach incrementally learns the condition under which the bug is observed by systematic negotiation with the user. In this process, LEARN2FIX generates alternative test inputs and sends some of those to the user for obtaining their labels. A limited query budget is assigned to the user for this task. A *query* is a *Yes/No* question: "When executing this alternative test input, the program under test produces the following output; is the bug observed?". Using the labelled test inputs, LEARN2FIX incrementally learns an *automatic bug oracle* to predict the user's response. A classification algorithm in machine learning is used for this task. Our key challenge is to maximise the oracle's accuracy in predicting the tests that expose the bug given a practical, small budget of queries. After learning the automatic oracle, an existing program repair tool attempts to repair the bug using the alternative tests that the user has labelled. Our experiments demonstrate that LEARN2FIX trains a sufficiently accurate automatic oracle with a reasonably low labelling effort (lt. 20 queries), and the oracles represented by *interpolation-based* classifiers produce more accurate predictions than those represented by *approximation-based* classifiers. Given the user-labelled test inputs, generated using the interpolation-based approach, the *GenProg* and *Angelix* automatic program repair tools produce patches that pass a much larger proportion of validation tests than the manually constructed test suites provided by the repair benchmark.

Index Terms—Automated Test Oracles, Semi-automatic Program Repair, Classification Algorithms, Active Machine Learning

1 INTRODUCTION

Finding and fixing software bugs is a significant concern in software development. The growing complexity of software systems has made this task challenging. Therefore, test-driven *Automated Program Repair* (APR) [1][2] has become an emerging research area. Recent advancements in APR technologies have been able to repair large software systems cost-effectively. However, the surveys of Gazzola et al. [1] and Le Goues et al. [2] indicate that there are some challenges to be addressed in APR. Among these challenges, *finding a test suite from a single bug-revealing input* for test-driven APR to produce an accurate patch is an important one.

Test-driven automated program repair techniques require a test suite (i.e. repair test suite): one or more failing tests exposing the bug that should be fixed and passing tests indicating the behaviour that should not be changed. Given a repair test suite, APR changes the buggy program to pass all the tests. Thus, the repair test suite has a significant impact on the quality of the repair. *Obtaining a repair test suite that leads to high-quality repair* is an interesting research problem in APR. The works of Yu et al. [3], Yang et al. [4] and Xiong et al. [5] have focused on this problem. All these approaches assume that a repair test suite is given in advance. Next, some test generation techniques are applied to augment the repair test suite in a manner improving the correctness of the patch. However, in most scenarios, the user reports a bug only with a single input exposing it (i.e., a single failing test case). These approaches cannot be directly used in such cases. Therefore, it is important to explore methods to obtain high-quality repair test suites beginning from a single input exposing a bug.

- *C. Geethal (charaka.kapugamawasangamagedon@monash.edu) is with Monash University, Australia*
- *M. Böhme is with the Max Planck Institute for Security and Privacy, Germany and Monash University, Australia.*
- *V.-T. Pham is with The University of Melbourne, Australia.*

Manuscript received August 14, 2023

To address the problem mentioned above, we envision a semi-automatic approach that generates a repair test suite by systematically learning, refining, and confirming the condition under which the bug is exposed from the human (user or developer). Our human-in-the-loop approach strategically asks the user: *"For the input $\vec{i}$, the program produces the output o; is the bug observed?"*. Even a user who has no experience with programming can answer this kind of question if they know the expected behaviour of the program. Based on the user's answers, our approach incrementally trains an *automatic bug oracle* that can predict the user's responses. The trained oracle can be used to ask the user more strategically. The key challenge in this setup is to maximise the oracle's accuracy under a limited number of queries to the user. The user-labelled test inputs are used to develop the repair test suite for the bug.

In this paper, we introduce LEARN2FIX, a technique that implements our approach for programs taking numeric inputs. LEARN2FIX begins with one failing test case of the buggy program. It uses *mutational fuzzing* [6] to generate alternative test cases, *active learning* [7] with a *classification algorithm* to train a classifier, and *automatic program repair* to fix the bug, using the human-labelled tests as the repair test suite. We expect that the test cases are sufficient to train a classifier as the oracle and repair the bug.

LEARN2FIX uses *mutational fuzzing* to generate new test cases in the "neighbourhood" of the given failing test. By exploring the neighbourhood, we can identify the "boundaries" of the bug and generate more failing tests in the "vicinity" of the given failing test. In mutational fuzzing, a test case is modelled as a sequence of numbers (i.e., bytes or integers), and new test inputs are generated by applying mutation operators at random positions in the sequence.

The test cases generated by mutational fuzzing should be human-labelled to be used in oracle learning. However, there are two main issues associated with this setup. Firstly, sending every generated input to the human would be impractical. Secondly, bugs are rarely exposed; hence, passing test cases are generated more frequently than failing test cases rendering the oracle learning subject to the *class imbalance problem* [8]: Given insufficient evidence about the bug (i.e. a relatively small number of failing test cases), the trained oracle might classify failing test cases with low accuracy. To address these issues, the most reasonable strategy would be to present to the human test cases having a higher probability of being labelled as *failing*.

As an automatic oracle is a binary classifier, a test case is classified either as "passing" or "failing". Thus, finding test cases with a higher *probability* of being "failing" is meaningless with respect to a single binary classifier whose prediction ability is unknown. Therefore, inspired by the approach of Holub et al. [9], LEARN2FIX creates an *unbiased committee of oracles* by applying slight changes to the training dataset that was used to train the original oracle. Given the test case t, LEARN2FIX asks each member of the committee to predict the label of t, and estimates the probability that t is failing as the proportion of members that classify t as "failing". If this probability is greater than a threshold $\hat{\theta} = 0.5$, then t is presented to the human. LEARN2FIX trains the automatic oracle applying a *classification algorithm* to the human-labelled test cases. Human labelling and oracle training happen incrementally in this process. Through this process, we expect to maximise human-labelling of failing tests and train highly accurate test oracles under a limited query budget.

In this paper, we extend our previous approach and results presented at ICST'20 [10] to investigate (i) the impact of choice of classifier to represent the automatic oracle, (ii) the impact of the choice of APR approach (search-based vs constrained-based), and (iii) the impact of mislabelling on the oracle quality and on the labelling effort. We also investigate (iv) the utility of LEARN2FIX in a pilot user study.

In terms of the choice of classifier, we distinguish interpolation-based and approximation-based approaches, where interpolation-based approaches *must* classify the training samples according to the given labels while approximation-based approaches are allowed to classify those differently to prevent overfitting. Previously, we used the *Incremental SMT Constraint Learner (INCAL)* [11] as the classification algorithm of LEARN2FIX and conducted experiments using only *GenProg* [12] as the automated program repair tool. In this extension, we evaluated which category of classifier representation from *interpolation-based* and *approximation-based* is most suitable for semantic bug automatic oracles. Interpolation and approximation are the two main classifier representations used in supervised machine learning. [13], [14].

In terms of APR approach, we evaluate a constraint-based based approach *Angelix* [15] in addition to the search-based, generate-and-validate approach *GenProg* [12] under each classification algorithm. This is to analyse whether LEARN2FIX auto-generated test suites can be applied to constraint-based repair. We chose *Angelix* [15], as it is a scalable multi-line APR technique that follows core principles of constraint-based repair.

Previously, we assumed that the user always provides the correct label for a test input generated by LEARN2FIX. However, in real situations, the user can incorrectly label test cases in oracle learning. Hence, we also analyse the impact of such noisy labels on oracle learning and automated program repair. Moreover, we conducted a pilot user study to study the applicability of LEARN2FIX in an actual human-in-the-loop environment.

Our experimental results demonstrate that

I. LEARN2FIX can train a sufficiently accurate automatic test oracle that can distinguish between passing and failing test under a reasonably low labelling effort. For the majority of subjects, the automatic oracles show more than 89% accuracy. LEARN2FIX uses a maximum of 20 queries to the user to train oracles with this much accuracy. Also, the user would mostly label failing tests, even though failing tests are rarely generated in the learning process (i.e., the probability of labelling a failing test is greater than the probability of generating a failing test). Thus, LEARN2FIX significantly reduces the effort of finding the failing tests of a bug.

II. LEARN2FIX shows better performance in terms of oracle quality and human labelling effort with interpolating binary classifiers than approximating binary classifiers. Firstly, LEARN2FIX can train more accurate automatic oracles with interpolation-based approaches than with approximation-based approaches. Secondly, interpolation based approaches reduce the effort of exploring failing tests than approximation-based approaches. LEARN2FIX can send most of generated failing tests to the human with interpolation-based approaches. Compared to our previous results [10], in which INCAL [11] was used as the classification algorithm, we find significant improvements in the oracle quality and probability of labelling failing tests when the *decision tree* representation is used. We believe that decision trees are able to more accurately capture the condition under which the bug is exposed than the other representations.

III. Both generate-and-validate and constraint-based program repair approaches produce more accurate patches with LEARN2FIX auto-generated repair test suites than with the manual repair test suites given by Codeflaws. LEARN2FIX auto-generated test suites through interpolation-based approaches show better performance in both APR approaches. In most repairable subjects, these test suites lead to producing patches that pass all the repair validation tests.

IV. Incorrectly labelled tests negatively affect the quality of the automatic oracles. Also, LEARN2FIX becomes unable to send failing tests to the user more frequently. Incorrectly labelled tests significantly affect the repair quality of the auto-generated test suites. The reason is that test-driven APR assumes that the test cases in the repair test suite contains correctly labelled test cases. In addition, the lack of failing tests in repair test suites further reduces the repair quality. Our pilot user study demonstrates that LEARN2FIX would work in a real human-in-the-loop environment.

In summary, the *main contributions* are as follows.

1) **Active Oracle Learning**. We introduce an active learning approach to derive an automatic oracle for a semantic bug by systematically interacting with the human. In this approach, we address the *class imbalance problem* by maximising human labelling of failing tests.
2) **Semi-Automatic Repair**. We introduce the first human-in-the-loop program repair technique, which systematically learns the condition under which the bug is exposed from the user before attempting to repair the program.
3) **Minimise Repair Overfitting**. Repair over-fitting is a key problem in automated program repair [2]. In this work, we propose a systematic approach to generate repair test suites that can mitigate repair overfitting.
4) **Evaluation and ablation**. We evaluate the quality of the automatic oracles and the patches generated by LEARN2FIX under approximation- and interpolation-based classifiers as automatic oracle, under different query budgets, and under APR tools implementing the search-based and constrained-based approach.
5) **Evaluation under mislabeling**. Generally, in automatic program repair as well as specifically in our human-in-the-loop approach to APR, we assume that the generated test inputs are correctly labelled. We analyze what happens if this assumption does not hold and how such incorrectly labelled test cases affect the active oracle learning process, oracle quality and semi-automatic automated program repair.

Reproducibility. To facilitate the reproduciblilty, we make our implementation of LEARN2FIX, our collection data, and scripts available at: https://github.com/charakageethal/learn2fix-journal-ext/.

2 Motivating Example

We demonstrate the existing challenges of automatic program repair using an example C program in Listing 1. This example is taken from an experiment conducted by Russ Williams [16]. In this experiment, 12 participants were asked to write programs to the *Triangle Classification Problem*, i.e., classifying triangles as *equilateral*, *isosceles*, *scalene*, and *invalid* given the lengths of the sides.

```
1  int f_steve_classify(int a,int b,int c){
2      if(a<=0 || b<=0 || c<=0)
3          return 4;   //Invalid
4      if(a<=c-b || b<=a-c || c<=b-a)
5          return 4;   //Invalid
6      if(a==b==c)     //BUG !
7          return 1;   //Equilateral
8      if(a==b || b==c || c==a)
9          return 2;   //Isosceles
10     return 3;
11 }
```

Listing 1: Buggy triangle classification program

`f_steve_classify` function takes 3 inputs that represent the lengths of the sides of a triangle and returns an integer where the return value
- 1 means it is equilateral (all sides equal in length)
- 2 means it is isosceles (exactly 2 equal sides)
- 3 means it is scalene (no equal sides)
- 4 means it is an invalid triangle

Functional bug. The C program in Listing 1 is Steve's implementation of triangle classification, which has a bug in Line 6. The programmer uses the C statement `a==b==c` (Line 6) instead of `a==b && b==c` to check whether the triangle is equilateral. Thus, given the input $t = (2, 2, 2)$, Line 6 evaluates it as follows.

`(2==2==2)` $\rightarrow$ `((2==2)==2)` $\rightarrow$ `((1)==2)` $\rightarrow$ `0`

The reason is that C represents boolean values `True` as 1 and `False` as 0, and thus `2==2` $\rightarrow$ `1` and `1==2` $\rightarrow$ `0`. Therefore, Listing 1 is incorrect for all equilateral triangles, except $\langle 1, 1, 1\rangle$, and for all isosceles triangles where `c=1`. For test input t, Listing 1 returns 2 (isosceles), while we expect it to return 1 (equilateral). This is a *Functional bug* or *Semantic bug*. Due to the difference between the actual and expected output, we identify t as a *failing test case*.

To identify t as a *failing test case*, we need to know the expected, correct output that Listing 1 should produce for t. Similarly, it is essential to know the expected, correct program behaviour of the *Program Under Test (PUT)* to detect functional bugs. Because of this reason, only the human (the developer or the user) can detect this category of bugs. However, bug detection through human involvement has many limitations. Therefore, developing techniques to learn automatic test oracles for functional bugs has a significant importance.

Automatic oracle. The program in Listing 1 fails for all inputs satisfying the following linear arithmetic constraint.

$$\begin{aligned} &[(a = b) \wedge (b = c) \wedge (a \neq 1) \wedge (o = 2)] \\ \vee &[(a = b) \wedge (c = 1) \wedge (a \neq 1) \wedge (o = 1)] \end{aligned} \tag{1}$$

where o = `f_steve_classify(a,b,c)` is the program output. We call this an automatic oracle for Steve's bug, as it identifies whether the given test case exposes Steve's bug.

Automatic Repair. Under a repair test suite containing a sufficient number of passing and failing test cases, an automated program repair tool [1] such as GenProg [12] or Semfix [17] would first identify Line 6 as the faulty statement. The reason is that most failing and least passing test cases actually execute this line (*Spectrum Based Fault Localization* [1]). Next, the repair tool would *repair* Line 6 such that all test cases are passing. However, we assume that there exists only one failing test case. Although Line 6 was detected as the faulty statement, the produce patch may be plausible but incorrect [18]. As an example, substituting "if" statement in Line 6 with `if(a==2)` would turn the test case $\langle 2, 2, 2\rangle$ into a passing one. However, this patch is overfitting and actually introduces a different bug.

Automatic oracle. If there was an automatic oracle, more failing test cases could be explored to generate a high-quality repair test suite. However, due to the necessity of expected program behaviour, only the user reporting the bug or the developers are the only oracles for functional bugs. Considering these limitations, we present an active learning approach to automatically derive automatic oracles similar to the one in Equation 1 for functional bugs.

3 LEARN2FIX Methodology

Given a buggy program ($\mathcal{P}$), we assume that there are one failing test case (f) and the human ($\mathcal{H}$) to answer whether a test is passing or failing. A test case t is of the form $t = \langle \vec{i}, o \rangle$, where $\vec{i}$ is a vector of input variable values, and $o = \mathcal{P}(\vec{i})$ is the output of $\mathcal{P}$ for $\vec{i}$. Also, we assume that $\vec{i}$ has a fixed length, and the human can answer at most L queries.

Algorithm 1 shows an overview of LEARN2FIX. The algorithm maintains two sets of test cases: T for all human labelled test cases and $T_{✗}$ for human labelled failing test cases ($T_{✗} \subseteq T$). Firstly, using the given failing input (f), LEARN2FIX trains an automatic oracle ($\mathcal{O}$) by a classification algorithm. As trained with a single failing input, $\mathcal{O}$ predicts everything as *failing* at this point.

More training data is required to improve the accuracy of the automatic oracle ($\mathcal{O}$). Hence, LEARN2FIX randomly selects a failing test case (f') from $T_{✗}$ and applies arithmetic mutations to generate a new test case (t) (Line 6 - Algorithm 1). t is presented to the human oracle ($\mathcal{H}$) for labelling if DECIDE2LABEL returns `true` (Line 7 - Algorithm 1). Next, t is added to T, and the automatic oracle ($\mathcal{O}$) is retrained with T (Line 13). if t is a *failing* test case, it is added to $\mathcal{T}_{✗}$ (Line 9). This process continues until the maximum number of labelling queries (L) is reached, or a timeout occurs.

Algorithm 1 LEARN2FIX Active Oracle Learning

```
Input: Buggy program ($\mathcal{P}$), Failing test case ($f = \langle \vec{i}, o \rangle$)
Input: Human oracle ($\mathcal{H}$), Maximum labelling queries ($L$)
 1: Failing test cases $T_{✗} \leftarrow \{f\}$
 2: Labelled test cases $T \leftarrow \{f\}$
 3: Automatic Oracle $\mathcal{O} \leftarrow$ TRAIN_CLASSIFIER($T$)
 4: while ($|T| < L$) and not timed out do
 5:     Failing test case $f' \leftarrow$ RANDOM_SELECT($T_{✗}$)
 6:     Generate test case $t \leftarrow$ MUTATE_FUZZ($f'$)
 7:     if DECIDE2LABEL($t, \mathcal{O}$) = true then
 8:         Human label $h = \mathcal{H}(t)$
 9:         if $h =$ fail then
10:             Failing test cases $T_{✗} \leftarrow T_{✗} \cup \{t\}$
11:         end if
12:         Labelled test cases $T \leftarrow T \cup \{t\}$
13:         Automatic Oracle $\mathcal{O} \leftarrow$ TRAIN_CLASSIFIER($T$)
14:     end if
15: end while
```

3.1 Generating More Failing Test Cases

A set of human labelled passing and failing test cases are required to train a classifier as an automatic test oracle. LEARN2FIX uses *mutational fuzzing* [6] for this task. Because of numeric inputs, LEARN2FIX applies *arithmetic mutations* [19] to f to generate new test cases. RANDOM_SELECT(T) in Algorithm 1 (Line 5) first randomly selects a seed failing test case $f' \in T_{✗}$. Then MUTATE_FUZZ

Algorithm 2 DECIDE2LABEL

```
Input: Unlabelled test case $t_?$, Automatic Oracle $\mathcal{O}$
Input: Committee Size $S$
 1: Let $T$ be training test cases that $\mathcal{O}$ has been trained
 2: Predicted label $\mathcal{L}_{\mathcal{O}} \leftarrow \mathcal{O}(t_?)$
 3: if $\mathcal{L}_{\mathcal{O}} =$ Failing then
 4:     return true
 5: else
 6:     votes = 0
 7:     for $i \leftarrow 1$ to $S$ do
 8:         Generated test case $t'_? =$ MUTATE_FUZZ($t_?$)
 9:         $t'_{✓} \leftarrow$ Assume that $t_?$ label as Passing
10:         $t'_{✗} \leftarrow$ Assume that $t_?$ label as Failing
11:         Hypothetical Oracle $\mathcal{O}_{✓} \leftarrow$ TRAIN_CLASSIFIER($T \cup \{t'_{✓}\}$)
12:         Hypothetical Oracle $\mathcal{O}_{✗} \leftarrow$ TRAIN_CLASSIFIER($T \cup \{t'_{✗}\}$)
13:         if $\mathcal{O}_{✓}(t_?) =$ Failing or $\mathcal{O}_{✗}(t_?) =$ Failing then
14:             votes $\leftarrow$ votes + 1
15:         end if
16:     end for
17:     $\widehat{\theta} = \frac{votes}{2 \times S}$
18:     if $\widehat{\theta} \geq 0.5$ then
19:         return true
20:     else
21:         return false
22:     end if
23: end if
```

applies arithmetic mutation operations (e.g. add one, subtract one, multiply by ten etc.) to f', which results in a new test case t. $t = \langle \vec{i'}, o' \rangle$ where $\vec{i'}$ is the input vector and $o' = \mathcal{P}(\vec{i'})$. This process generates new test cases in the "vicinity" of f.

The ability of *mutational fuzzing* to generate more test cases in the neighbourhood of a failing helps to collect more evidence of the location and behaviour of the bug. This approach has been proven to be successful in the coverage-based, mutational fuzzer *American Fuzzy Lop (AFL)* [20], which generates more crashing inputs by mutating a seed crashing input. The neighbourhood test cases generated given by mutations demonstrate how the program's behaviour changes from buggy to correct and vice versa, under small changes to the input. Also, mutational fuzzing has a higher probability of generating failing test cases compared to *generational fuzzing* [21].

Example: $t_{✗} = \langle \langle 2, 2, 2 \rangle, 2 \rangle$ is a failing test case of the motivating example in Listing 1. For illustration, assume that for each position a in $\vec{i}$, we employ one of the three mutation operators uniformly chosen at random: $\vec{i'}[a] = \vec{i}[a]$, $\vec{i'}[a] = \vec{i}[a] + 1$, or $\vec{i'}[a] = \vec{i}[a] - 1$. The following test cases are generated when actually running the mutational fuzzer on $t_{✗}$.

$$\begin{array}{ll}
\langle \langle 2,2,1 \rangle, 1 \rangle_? & \langle \langle 1,3,3 \rangle, 2 \rangle_? \\
\langle \langle 1,3,2 \rangle, 4 \rangle_? & \langle \langle 3,3,1 \rangle, 1 \rangle_? \\
\langle \langle 2,1,3 \rangle, 4 \rangle_? & \langle \langle 3,3,3 \rangle, 2 \rangle_? \\
\langle \langle 2,1,1 \rangle, 4 \rangle_? & \langle \langle 1,2,3 \rangle, 4 \rangle_? \\
\langle \langle 3,2,2 \rangle, 2 \rangle_? & \langle \langle 2,3,2 \rangle, 2 \rangle_?
\end{array}$$

Three out of ten cases generated above expose Steve's error (if labelled by $\mathcal{H}$), i.e., $\langle\langle 2,2,1\rangle,1\rangle$, $\langle\langle 3,3,1\rangle,1\rangle$, and $\langle\langle 3,3,3\rangle,2\rangle$.

In contrast to mutational fuzzing, generational fuzzing generates random inputs that adhere to the input format of the system under test [21]. The inputs (a,b and c) in Listing 1 can take any integer values specified by C programming language. Assume that we randomly generate three integers in the range $[-2^{63}, 2^{63}-1]$. In this approach, the probability of finding a test case representing an isosceles triangle with $c = 1$ or an equilateral triangle with $c \neq 1$ is extremely low. Thus, mutational fuzzing has a higher probability of generating *failing test cases* than generational fuzzing.

3.2 Training a Classifier as a Test Oracle

To compute the automatic oracle, LEARN2FIX trains a binary classifier based on a human labelled training data set. The function TRAIN_CLASSIFIER uses the same classification algorithm in both Algorithm 1 and Algorithm 2. We consider the input ($\vec{i}$) and the corresponding program output values (o) of a test case as the *features* for the classification algorithm. We consider the labels *Passing* and *Failing* as the two classes to be predicted. The function TRAIN_CLASSIFIER uses test cases labelled by the human (T) to train a binary classifier as the automatic oracle ($\mathcal{O}$). Given a test case, an automatic oracle ($\mathcal{O}$) predicts the label based on the input ($\vec{i}$) and corresponding buggy program output (o).

Usually, a classification algorithm requires at least one data point from each class. However, human-labelled test suites (training test suites) containing only failing tests can be generated in oracle learning. If so, we assume that TRAIN_CLASSIFIER returns a classifier that predicts every test case as *failing*.

There are many binary classification algorithms in machine learning to work with numeric data. Based on the classifier representation, classification algorithms can be divided into two categories: *interpolating* [13] and *approximation* [14].

Interpolation-based classification algorithms explore a model that exactly fits the training data. Some classification algorithms under this category infer a set of constraints fitting the given data points. *AdaBoost* and *Decision Tree* are examples of such algorithms. The work of Braga et al. [22] uses *AdaBoost* algorithm to develop test oracles. Also, the survey paper of Briand et al. [23] suggests that *decision trees* are effective in modelling the failure condition of a bug. Due to these reasons, we evaluated the performance of *AdaBoost* and *Decision Tree* classification algorithms with LEARN2FIX. In addition, we selected *Incremental SMT Constraint Learner* (INCAL) [11], which generates interpolation binary classifiers as Satisfiability Modulo Theory (SMT) [24] formula. *Symbolic Execution* [25] uses SMT constraints to group the inputs that exercise a particular path. Thus, SMT formula can be used to group the failing and passing inputs of a bug. For this reason, we selected INCAL [11] as a classification algorithm for our experiments.

Approximation-based classification algorithms approximate a model for the training data as minimizing the empirical error. Thus, the model does not exactly fit the training data. *Artificial Neural Networks* belong to this category. The work of Jin et al. [26] uses two artificial neural network setups to generate automatic test oracles. One setup has two hidden layer with 20 and 5 neurons (MLP(20,5)). The other setup has only one hidden layer with 20 neurons (MLP(20)). We selected these neural network configurations for our experiments. In addition, we chose *Support Vector Machine* and *Naïve Bayes* under approximation-based classification algorithms. *Support Vector Machine* is an algorithm that can be used in high-dimensional or infinite-dimensional space [27]. *Naïve Bayes* is based on the Bayes theorem and able to learn an accurate classifier with relatively less training data [27]. These two algorithms have been applied in different domains; however, their applicability to test oracle automation has not been explored.

The selected set of classification algorithms is as follows.

i. Incremental SMT Constraint Learner (INCAL)
ii. Decision Tree (DT)
iii. AdaBoost (ADB)
iv. Support Vector Machine (SVM)
v. Naïve Bayes (NB)
vi. Neural Networks / Multi-Layer perceptrons (MLP)

We experimentally evaluate the performance of these algorithms to know which category of classifier representation (interpolation or approximation) is most suitable for LEARN2FIX. Moreover, we explore the best-performing classifier representation with LEARN2FIX.

3.3 Maximising the Probability of Labelling Failing Test Cases

As the minority class is *failing*, LEARN2FIX improves the classifier's ability to identify *failing* test cases, using the limited human queries. For this purpose, LEARN2FIX maximises the probability of labelling failing test cases in oracle learning. This strategy helps to address the *class imbalance problem*. To maximise the probability of labelling failing test cases, LEARN2FIX selects test cases with higher failure likelihood. Algorithm 2 (DECIDE2LABEL) describes this process. This method has been influenced by the work of Holub et al. [9]. Following Holub's method, DECIDE2LABEL estimates the failure likelihood based on the current status of the automatic oracle ($\mathcal{O}$).

The key concept in Holub's method is to select the *Most Informative Unlabelled Point (MIUP)* for labelling based on the current status of the classifier. Holub's method considers the data point with the *Minimum Expected Entropy (MEE)* [9] as the MIUP. In finding the data point with MEE, Holub's method estimates the *look-ahead probability* of each class based on a *committee of classifiers* with hypothesized labels [9].

The DECIDE2LABEL-algorithm sends test cases predicted as *failing* by the automatic oracle being trained ($\mathcal{O}$) for human labelling. If the given test case ($t_?$) is actually *failing*, human labelling of $t_?$ allows $\mathcal{O}$ to learn more about the failure. If $t_?$ is actually a *passing* test case, it implies that $\mathcal{O}$ has not been trained correctly. In this case, LEARN2FIX rectifies $\mathcal{O}$ by human labelling of $t_?$ and using it in training.

If $\mathcal{O}$ predicts $t_?$ as *passing*, the DECIDE2LABEL-algorithm calculates the probability that $\mathcal{O}$ predicts $t_?$ as *failing*. Intuitively, there is an equal probability of classifying a test case into either class. LEARN2FIX estimates the probability that $\mathcal{O}$ predicts $t_?$ as *failing* one-step ahead. Following Holub's look-ahead probability estimation method [9], the DECIDE2LABEL-algorithm constructs a committee of automatic oracles (Line 7-16) for this task.

In creating the oracle committee, first, the DECIDE2LABEL-algorithm generates a new test case ($t'_?$) by applying mutational fuzzing to $t_?$. The new test input $t'_?$ is hypothetically labelled as *passing* ($t'_{\checkmark}$) (Line 9). Then, a new hypothetical oracle ($\mathcal{O}_{\checkmark}$) is trained with the training set $T \cup \{t'_{\checkmark}\}$ (Line 11, T: The initial training set of $\mathcal{O}$). The same test case is hypothetically labelled as *failing* ($t'_{\times}$)(Line 10). Another hypothetical oracle ($\mathcal{O}_{\times}$) is trained with the training set $T \cup \{t'_{\times}\}$ (Line 12). The DECIDE2LABEL-algorithm generates 2 hypothetical oracles for a newly generated test case. Thus, in S fuzzing iterations, a committee containing $2 \times S$ automatic oracles is generated. Each hypothetical oracle created by adding a hypothetically labelled test case to the initial training (T) set demonstrates a possible status of the automatic oracle ($\mathcal{O}$) one step ahead. As each newly generated test case ($t_?$) is hypothetically labelled as both *passing* and *failing* contributing to two different hypothetical oracles, the oracle committee overall is *unbiased*.

Finally, the unlabelled test case $t_?$ is presented to the oracle committee, and the occurrences that $t_?$ is predicted as *failing*, i.e., *fail_votes*, are counted. (Line 11-12). As there are $2 \times S$ oracles in the committee, the probability of labelling $t_?$ as *failing* is estimated by $\widehat{\theta} = \frac{\mathit{fail\_votes}}{2 \times S}$. As the oracles in the committee are some possible future states of $\mathcal{O}$, $\widehat{\theta}$ is a *look-ahead estimation* of the probability of *failing*. The DECIDE2LABEL-algorithm considers that test cases with $\widehat{\theta} \geq 0.5$ have higher failure likelihood and sends those for human labelling (Line 18). According to the oracle committee, if $t_?$ has a higher failure probability, it implies that the automatic oracle ($\mathcal{O}$) has not been trained adequately to identify the failing test cases. Thus, labelling such test cases and using them in training rectify the automatic oracle ($\mathcal{O}$).

3.4 Automatic Program Repair

Algorithm 1 returns a human-labelled test suite T, containing both passing and failing tests, as an additional outcome of the oracle learning. The labelled test suite T is used as a repair test suite with a *test-driven* automated program repair (APR) tool [12], [15] to repair the buggy program ($\mathcal{P}$).

To generate a fix for the given buggy program, test-driven APR techniques use a test suite containing *passing* and *failing* test cases. The failing tests exercise the bug to be fixed, while the passing tests indicate the behaviour that should not be changed. This test suite is known as *repair test suite*. Given the repair test suite, the APR technique changes the buggy program to pass all the test cases. According to Le Goues et al. [2], *Heuristic repair* and *Constrained-based repair* are the two main categories of test-driven automated program repair techniques. In common, these two categories use the repair test suite for *fault localization* [28], i.e., finding the code locations that are likely to be buggy.

Heuristic / generate-and-validate techniques iteratively generate and validate repair candidates, modifying the given buggy program. To generate repair candidates, these techniques apply syntactical modifications to the given buggy program. The abstract syntax tree (AST) representation of the buggy program is used in this process. To reduce the search space and guide the syntactical modifications, heuristic repair techniques use the information obtained in the fault localization. After a repair candidate is generated, the validation step calculates the number of tests in the repair test suite passed by the candidate. The generate and validate process continues until a repair candidate passing all the tests in the repair test suite is found. *GenProg* [12] is a popular heuristic repair technique that uses an extended form of *genetic programming* [29] to generate repair candidates.

Constraint-based repair techniques explore a repair constraint that the patched program should satisfy, rather than modifying the program to generate patches [1], [2]. The patch (typically a code segment) to be generated is considered as an unknown function. The fault localization indicates where the patch should be placed. The properties about the unknown function are extracted through symbolic execution [25] or other methods; these properties constitute the repair constraint. A patch for the bug is explored by finding a solution to the repair constraint. This is usually achieved by search or constraint solving. *Angelix* [15] is an example of constraint-based repair technique.

4 Experimental Setup

We empirically evaluated the different aspects of LEARN2FIX. Firstly, we evaluated the *quality of the automatic oracles* and the *human effort* involved in the learning process. Secondly, we analysed how these two factors vary across different classifier representations. Thirdly, we examined the applicability of the test suites generated in oracle learning (T - Algorithm 1) to automated program repair. Finally, we analysed the impact of mislabelled tests on the oracle quality, human effort and automated program repair. The research questions in Section 4.1 were used to guide these experimental tasks.

4.1 Research Questions

RQ.1. **(Oracle Quality)** How accurate are automatic oracles trained by LEARN2FIX in classifying test cases in the repair benchmark?

RQ.2. **(Labelling Effort)** What is the proportion of generated test cases that are sent to the human oracle for labelling? Does the probability of sending failing test cases indeed increase versus a random choice of test cases?

RQ.3. **(Oracle Representation)** Which category of classifier representation (interpolation or approximation) works better with LEARN2FIX?

RQ.4. **(Patch Quality)** How does the quality of patches produced through LEARN2FIX's automatically generated test suites compare to the quality of patches produced through the manually constructed test suites given by the benchmark? How many subjects can be repaired, and what is the proportion of validation test cases that the patched program passes? How does the patch quality vary across different classifier representations?

RQ.5. **(Impact of Noisy Labels)** How do incorrectly (noisy) labelled test cases affect the oracle quality, human effort and patch quality?

4.2 Experimental Subjects

To evaluate LEARN2FIX and answer the research questions, we selected 552 programs from *Codeflaws* [30] benchmark according to the following criteria.

1) There should be a sufficiently large number of programs that are algorithmically complex.
2) There should be a diverse set of real world defects that cause *functional bugs*, i.e., programs produce incorrect or unexpected output for certain inputs. There should be one functional bug for each subject.
3) For each subject, there should be a *golden version*, i.e., a program that produces the expected, correct output for an input. For a given input we simulate the Human oracle ($\mathcal{H}$) by comparing the subject's (buggy program's) output with its golden version's output. If both outputs are different, the human label of the test case is considered as *failing*.
4) For each subject, there should be a *manually constructed* and labelled *repair test suite* and *repair validation test suite*. We use both test suites combined to evaluate oracle quality. The repair test suite is used to generate a patch with the automated program repair tool, and the repair validation test suite is used to evaluate the patch.
5) For each subject, there should be at least one failing test case in the repair test suite, i.e., a test input for which the buggy program and its golden version produce different outputs. Otherwise, LEARN2FIX cannot be started.
6) For each subject, there should be test inputs having a constant number of numeric values. For each such test input, the program should produce a numeric output. Otherwise, the classification algorithms cannot be applied to learn the automatic oracle ($\mathcal{O}$).

Codeflaws consists of 3902 buggy programs that belong to 40 real-world defect classes. These programs have been written in C programming language and extracted from the Codeforces online database. For each buggy program, there is a manually constructed and labelled *repair test suite* and *repair validation test suite*. In this benchmark, the repair validation test suites are named *held-out test suites*. Tan et al., the authors of Codeflaws, claim that ”to our best knowledge, in automatic program repair evaluation, our benchmark has the largest number of real defects obtained from the largest number of subject programs to date” [30].

The selected subjects from Codeflaws for the experiments belong to 34 defect classes. (Table 1). Each program takes a fixed number of numeric inputs and returns a numeric output. Figure 2 shows the distributions of the manually constructed test cases, given by Codeflaws, of the selected subjects. In most subjects, there are more passing test cases than failing test cases (Figure 2b).

We ignored IntroClass and ManyBugs benchmarks [31], as those do not satisfy our selection criteria. ManyBugs contains programs taking complex and non-numeric inputs, which violates our sixth criterion. The programs taking numeric inputs in IntroClass have very simple functions (e.g. return the smallest of three numbers), which does not satisfy the first criterion.

4.3 Automated Program Repair Tools

We selected *GenProg* [12] and *Angelix* [15] as the automated program repair (APR) tools in the experiments. Several studies related to APR have considered these two tools as the state-of-art APR tools (e.g. Le Goues et al. [2], Yi et al. [32], Motwani [33] and Le et al. [34]). *GenProg* is a *Heuristic / Generate-and-validate* repair tool, whereas *Angelix* is a *Constraint-based* repair tool [2]. These APR tools have shown their capability to repair large programs cost-effectively. Also, these both *GenProg* and *Angelix* are already set up with Codeflaws benchmark.

4.4 Setup and Evaluation

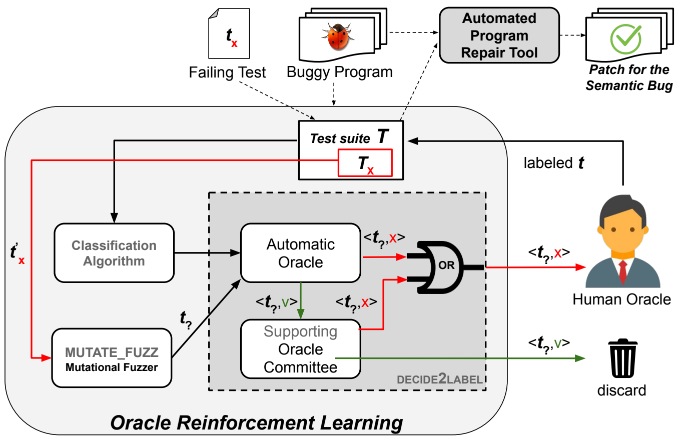


Fig. 1: Workflow of LEARN2FIX

First, we select one from the algorithms listed in Section 3.2 as the classification algorithm of LEARN2FIX. For each program subject, we randomly select a failing test case from the *repair test suite* as the input to LEARN2FIX. After LEARN2FIX generates the automatic oracle, we apply it to predict the labels of the test cases in the manually constructed test suite (i.e., *repair test suite* + *held-out test suite*) given by the benchmark.

A human-labelled test suite (T-Algorithm 1), including both *passing* and *failing* test cases, is generated in this process. The failing test cases in T do not contain the expected, correct outputs for the inputs. We replace the output of each failing test case with its expected output to convert T to a repair test suite. We call this repair test suite *auto-generated* repair test suite. Then, we attempt to repair the program with the auto-generated repair test suite and the manually constructed repair test suite, given by the benchmark, separately using an APR technique in Section 4.3. Under each test suite, if the APR technique generates a patch, we count the number of tests in the held-out test suite (validation test suite) passed on the patched program.

Figure 1 shows the detailed workflow of this process. We repeat this process for each classification algorithm (Section 3.2) and for each automated program repair technique (Section 4.3). The results of the *best-performing* classification algorithm in oracle learning are used to answer **RQ.1.** and

Defect Class	Description	Example	No. of Subjects
DCCR	Replace Constant with variable/constant	`- for(i=n+1;i<=90;i++)` `+ for(i=n+1;i<=100;i++)`	71
OILN	Tighten condition or loosen condition	`- if(t%2==0)` `+ if(t%2==0 && t=2)`	64
ORRN	Replace relational operator	`- if(sum>n)` `+ if(sum>=n)`	59
HIMS	Insert multiple non-branch statements	`+ frepoen("input.txt","r",stdin);` `+ freopen("input.txt","w",stdout);`	52
HOTH	Other higher order defect classes	`- scanf("%s",h);` `+ for(i=0;i<71;i++)` `+ scanf("%c",&h[i]);`	48
OAIS	Insert / Delete arithmetic operator	`- max += days%2` `+ max += (days%7)%2`	47
STYP	Replace variable declaration type	`- int a;` `+ long a;`	37
DRVA	Replace a read variable with a variable/constant	`- for(i=0;i<l;i++)` `+ for(i=0;i<m;i++)`	28
SMOV	Move statement	`- scanf("%d",&i);` `scanf("%s",&a);` `+ scanf("%d",&i);`	19
DRWV	Replace a write variable with a variable	`- b=0;` `+ a=0;`	17

TABLE 1: Top 10 defect classes (out of 34) of the 552 Codeflaws subjects selected for the experiments

RQ.2.. To answer **RQ.3.**, we use the data collected under all the classification algorithms. The data collected under the program repair experiments are used to answer **RQ.4.**.

To answer **RQ.5.**, we consider 5%, 10% and 20% of the allocated human queries (i.e., maximum labelling effort-L) are incorrectly answered. The incorrectly labelled test cases are introduced at random positions in active oracle learning. Under each noise level above, we repeat the experiments related to **RQ.1.**, **RQ.2.** and **RQ.4.**.

For our experiments, we fixed the following values.

- *Timeouts*. In each subject, we allocated 10 minutes per each for oracle learning (Algorithm 1) and auto-generating a patch.
- *Committee Size*. We set the size of the oracle committee to 20 members (i.e., $S = 10$ in Algorithm 2).
- *Maximum Labelling Effort*. We set the maximum labelling effort to the human oracle ($\mathcal{O}$) to 20. (i.e., $L = 20$ in Algorithm 1)

Related to **RQ.1.**, comparing the predicted labels by the automatic oracle ($\mathcal{O}$) with the actual labels of the test cases, we calculated *Accuracy* (Equation 2) *Recall - Failing* (Equation 3), *Recall - Passing* (Equation 3), *Precision-Failing* (Equation 5) and *Precision-Passing* (Equation 6). In some experiments, we considered *F-scores* (Equation 7) of passing and failing tests. It can indicate the variations of both precision and recall.

$$\textit{Accuracy} = \frac{\textit{Number of correctly predicted tests}}{\textit{Number of test inputs in the test suite}} \tag{2}$$

$$\textit{Recall-Failing} = \frac{\begin{array}{c}\textit{Number of correctly predicted}\\ \textit{failing tests}\end{array}}{\begin{array}{c}\textit{Number of failing inputs}\\ \textit{in the test suite}\end{array}} \tag{3}$$

$$\textit{Recall-Passing} = \frac{\begin{array}{c}\textit{Number of correctly predicted}\\ \textit{passing tests}\end{array}}{\begin{array}{c}\textit{Number of passing tests}\\ \textit{in the test suite}\end{array}} \tag{4}$$

$$\textit{Precision-Failing} = \frac{\begin{array}{c}\textit{Number of correctly predicted}\\ \textit{failing tests}\end{array}}{\begin{array}{c}\textit{Total number of tests}\\ \textit{predicted as failing}\end{array}} \tag{5}$$

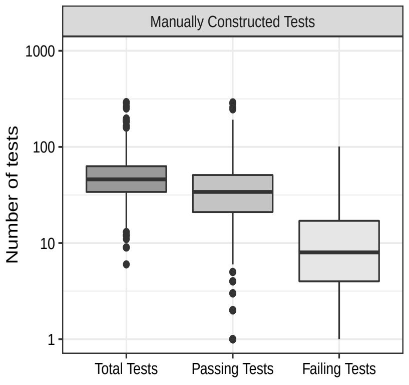
$$\textit{Precision-Passing} = \frac{\begin{array}{c}\textit{Number of correctly predicted}\\ \textit{passing tests}\end{array}}{\begin{array}{c}\textit{Total number of tests}\\ \textit{predicted as passing}\end{array}} \tag{6}$$

$$\textit{F-Score} = \frac{2 \times \textit{Precision} \times \textit{Recall}}{\textit{Precision} + \textit{Recall}} \tag{7}$$

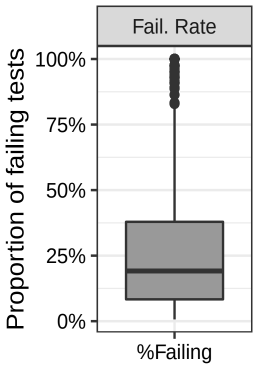
We have identified that the labelled test suites of most selected subjects from Codeflaws have more passing test cases and fewer failing test cases. Thus, the *Class Imbalance Problem* [8] impacts the evaluation. For this reason, *Accuracy* is not a good metric of oracle quality. For example, an oracle predicting everything as passing would be 90% accurate for a test suite containing 90% of passing test cases. Therefore, we report *Accuracy*, *Recall-Failing*, *Recall-Passing*, *Precision-Failing* and *Precision-Passing*. Also, these metrics help to evaluate the effectiveness of the techniques that LEARN2FIX uses to deal with the class imbalance problem (Section 3.3 & Section 3.3).

To address **RQ.2.**, we measured the following.

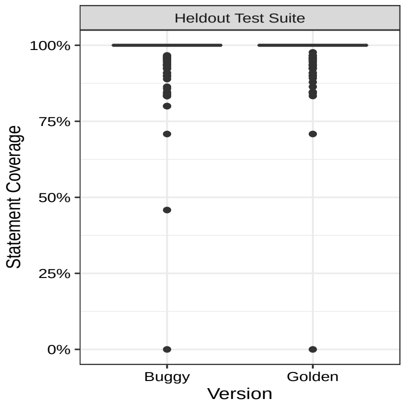
i. The proportion of generated tests that are labelled (Equation 8).
ii. The proportion of failing tests that are labelled from the generated(Equation 9)



(a) Distribution of tests



(b) Percentage of failing tests



(c) Statement coverage of held-out test suites

Fig. 2: Manually constructed test suites of Codeflaws. *On the left (a)*: Distribution of passing and failing tests and the number of total tests. *In the middle (b)*: Failing tests to passing tests ratio. *On the right (c)*: Statement coverage of held-out test suites in buggy and golden versions of the selected subjects

iii. The probability to generate a failing test (Equation 10)
iv. The probability to label a failing test (Equation 11).

$$\begin{array}{c}\textit{Proportion of generated}\\ \textit{tests labelled}\end{array} = \frac{\textit{Number of tests labelled}}{\textit{Number of tests generated}} \quad (8)$$

$$\begin{array}{c}\textit{Proportion of failing}\\ \textit{tests labelled}\end{array} = \frac{\textit{Number of failing tests labelled}}{\textit{Number of failing tests generated}} \quad (9)$$

$$\begin{array}{c}\textit{Probability to generate}\\ \textit{a failing test case}\end{array} = \frac{\textit{Number of failing tests generated}}{\textit{Total number of tests generated}} \quad (10)$$

$$\begin{array}{c}\textit{Probability to label}\\ \textit{a failing test case}\end{array} = \frac{\textit{Number of labelled failing tests}}{\textit{Total number of labelled tests}} \quad (11)$$

One objective of LEARN2FIX is to reduce the number of labelling queries to the human while maximising the probability of human labelling a failing test case. Equation 10 indicates the probability of generating a failing test case in oracle learning. This is also the probability that the human would find a failing test only by mutational fuzzing and without LEARN2FIX. We compare this with the *probability of labelling a failing test case* (Equation 11). If the probability of labelling a failing test case is greater than the probability of generating a failing test case, the human needs less effort than usual to explore failing test cases. Equation 9 assesses the capability of LEARN2FIX to select failing test cases given by mutational fuzzing. By comparing this with the *proportion of generated tests that are labelled*, we can identify how effectively LEARN2FIX utilizes the given query budget to explore failing tests.

For exploring answer to **RQ.3.**, we computed the same metrics used in **RQ.1.** and **RQ.2.** for each classification algorithm.

Regarding **RQ.4.**, we measured the following for the manual test suite given by the benchmark and the auto-generated test suite by LEARN2FIX.

i. *Repairability*: The proportion of the subject that can be repaired by the APR tool (Equation 12)
ii. *Validation Score*: The proportion of *validation test cases* that the patched program passes (Equation 13).

$$\textit{Repairability} = \frac{\begin{array}{c}\textit{Number of subjects that were}\\ \textit{successfully repaired}\end{array}}{\textit{Total number of subjects}} \quad (12)$$

$$\textit{Validation Score} = \frac{\begin{array}{c}\textit{Number of repair validation tests passed}\\ \textit{on the patched program}\end{array}}{\begin{array}{c}\textit{Total number of tests in the}\\ \textit{repair validation test suite}\end{array}} \quad (13)$$

In addition to these metrics, we computed the statement coverage and composition (i.e., total number of tests, passing and failing tests) of the manual and auto-generated test suites to support our analysis. In each APR technique, we examined the defect categories repaired by using the manual and auto-generated test suites with all the classification algorithms.

We measured the *repairability* (Equation 12), as APR tools fail to produce repairs with some repair test suites within an allocated time. If the APR tool generates a patch, the *validation score* (Equation 13) can be calculated. Regarding a program subject, we used the *held-out test suite* as the repair validation test suite. According to Tan et al. [30], the held-out test suites in Codeflaws are large enough for evaluating patch correctness. In addition, we observed that the heldout-test suite achieved 100% *statement coverage* [35] in the golden versions of most subjects (Figure 2c-Golden box). It implies that the held-out test suite can exercise all the expected correct behaviours of a subject. Therefore, the manually created held-out test suites are suitable for evaluating patch correctness in APR. If a patch can achieve 100% validation score, it implies that the patch is accurate and non-overfitting.

Similar to our study, Motwani et al. [36] and Brun et al. [37] use separate repair validation test suites to evaluate the correctness of a patch. These works also use the metrics *repairability* and *validation score*. Motwani's study claims that using a separate repair validation test suite is more objective than manually evaluating patch correctness and reproducible in a fully-automated manner.

To mitigate the impact of randomness and to gain statistical power for the experimental results, we repeat each experiment 30 times.

5 Experimental Results

5.1 RQ.1.: Oracle Quality

We investigate the quality of the automatic oracle under the best-performing oracle representation (i.e., *Decision Tree*).[1] Figure 3 shows the results for the automatic oracles trained by LEARN2FIX as average over 30 runs distributed over the different subjects.

> Under a maximum of 20 queries to the user, for the majority of subjects, the automatic oracles trained by LEARN2FIX are able to accurately predict the labels of more than 89% of the manually labelled tests given by the benchmark ("Overall"; Fig. 3). Even though LEARN2FIX has seen only one failing test, the automatic oracle correctly identifies more than 80% of the failing tests in most subjects ("Failing-Recall"). In addition, the precision and recall of passing test cases are more than 90% for the median subject ("Failing-Precision").

Figure 3 shows the results for a fixed budget of 20 queries to the user. The median values of all the metrics are above 75%. Thus, LEARN2FIX is able to train automatic oracles that accurately distinguish the passing and failing tests of the majority of subjects, using just one failing test. The higher median values in failing-recall and failing-precision suggest that LEARN2FIX is able to successfully deal with the *class imbalance problem*. The results suggest that LEARN2FIX can train highly accurate test oracles getting the maximum use of the available human queries. The number of queries (20) is reasonable to the human, as these are yes/no questions.

Figure 4 shows how oracle quality is impacted if we change the allocated query budget (i.e., the maximum number of labelling queries) to the human.

> As the maximum number of labelling queries increases, the oracle quality is improved as well.

When the maximum number of queries to the human increases, LEARN2FIX can obtain more labelled test cases for oracle learning. Consequently, LEARN2FIX can learn the failure condition of a bug more accurately. Hence, the overall accuracy increases (Figure 4-Overall). Also, the ability to correctly distinguish between passing and failing test cases is improved. The increases in F-score of passing and failing test cases imply this fact. When there are fewer labelled test cases, the decision tree algorithm over-approximates the failure condition. As an example, when the maximum labelling effort is 5, the median of failing-recall is above 75%, while the median of failing-precision is below 50%.

5.2 RQ.2.: Labelling Effort

We investigate the labelling effort under the best-performing oracle representation.[1] The boxplots in Figure 5a show the proportion of generated (left) and failing (right) tests that are labelled. In addition, Figure 5b shows the distribution of the probability to generate (left) and label (right) label a failing test.

> Despite choosing only a small proportion of generated test inputs for labelling, LEARN2FIX is effective at sending mostly failing test inputs for labelling and successfully tackles the class imbalance problem.

Under a maximum of 20 queries to the user, Figure 5a shows that for the median subject, despite sending less than 25% of generated test inputs for labelling, LEARN2FIX asks for the label of more than 75% of generated failing test inputs. In contrast, a random selection of generated inputs would send substantially less failing inputs for labelling, indicating a substantial reduction in labelling effort by LEARN2FIX. Figure 5b shows that for the median subject despite a probability of less than 25% of generating failing test inputs, the probability that a test input that is sent for labelling is failing is more than 60%. As failing test inputs are in the minority class during generation, this means that LEARN2FIX is effective at tackling the class imbalance problem during training.

Figure 6 shows how labelling effort is impacted if we change the allocated query budget (i.e., the maximum number of labelling queries) to the human.

> As the query budget increases, the proportion of generated tests sent for labelling decreases (Figure 6a-left). Nevertheless, the median percentage of failing tests sent for labelling is above 70% across all query budgets (Figure 6a-right). The probability of generating a failing test case does not significantly change (almost the same) across query budgets (Figure 6b-left). Nonetheless, the probability of labelling a failing test case increases as more queries are sent for labelling (Figure 6b-right).

We previously observed that the accuracy of the automatic oracle increases as more labelling queries are sent (Figure 4). As the accuracy of the automatic oracle increases, the DECIDE2LABEL algorithm can select more failing test cases for labelling, thus increasing the percentage of labelled failing test cases (Figure 6a-right). Due to the same reason, most of generated passing test cases (the majority class) are not sent for labelling. This is the reason for the decreases in the percentage of generated tests sent for human labelling (Figure 6a-left). All these facts lead to increasing the probability of labelling a failing test case (Figure 6b-right). The increasing probability of labelling a failing test case implies

1. We report results for other classifier representations in Section 5.3. According to the results, decision tree is one of the best-performing classifier representations with LEARN2FIX.

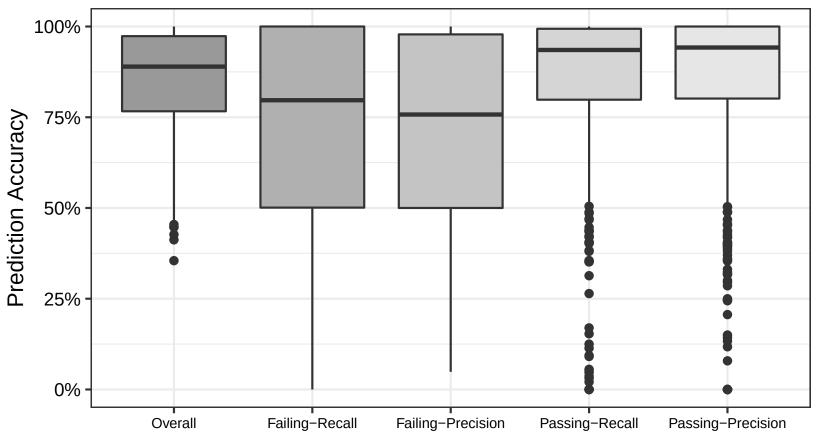


Fig. 3: LEARN2FIX Oracle quality under the *Decision Tree* algorithm

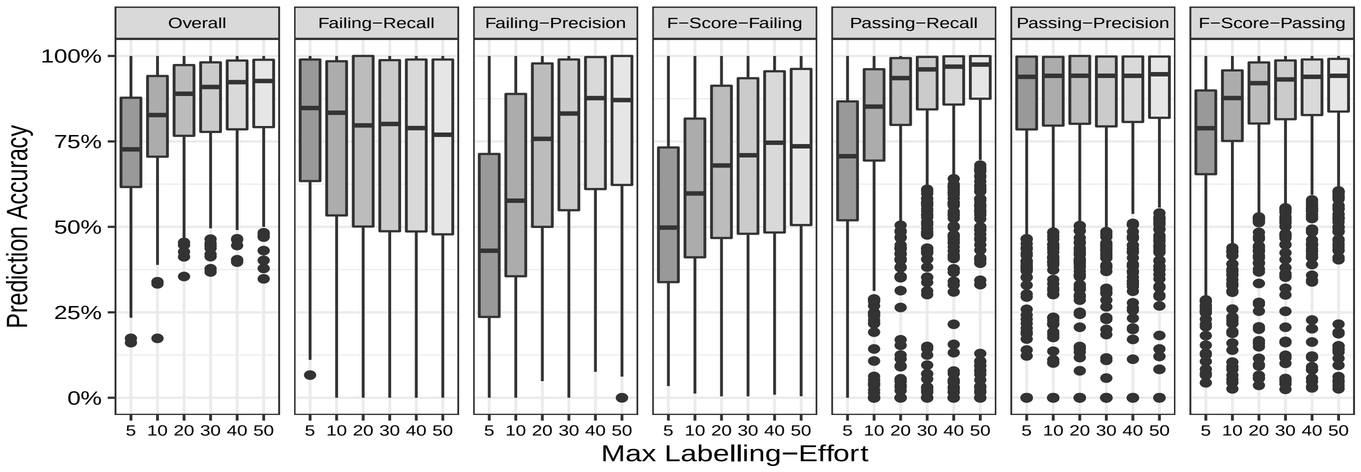
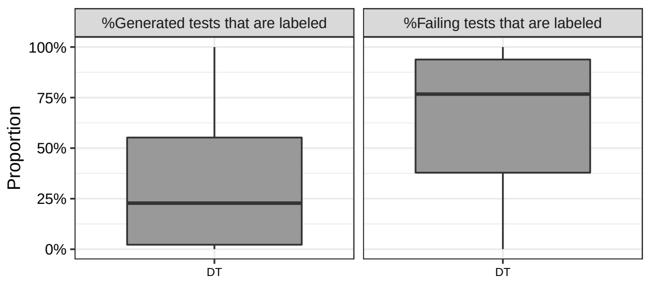
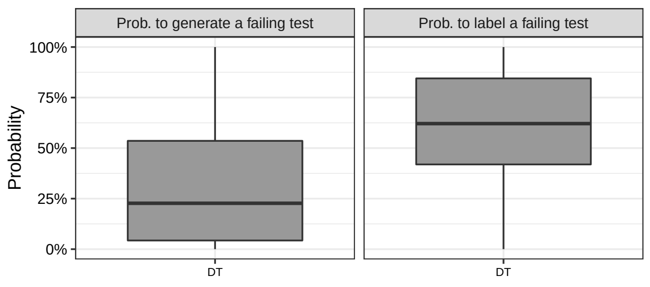


Fig. 4: Variations of oracle quality under the maximum number of queries 5,10,20,30,40 and 50

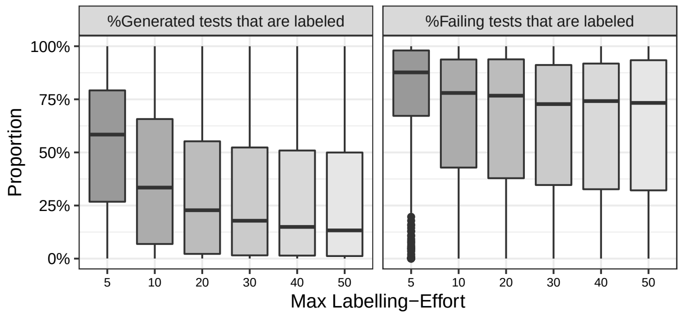


(a) Proportion of generated (left) / failing tests that are labelled (right)

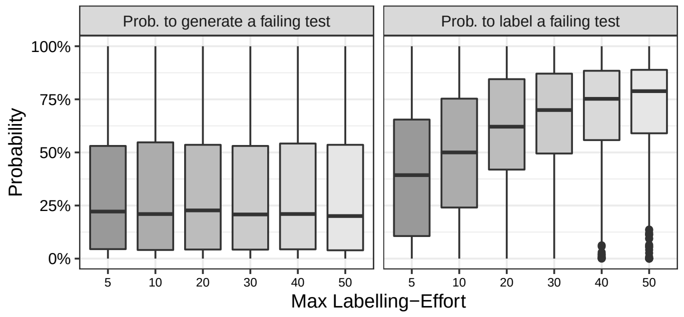


(b) Probability to generate (left) / label a failing test (right)

Fig. 5: LEARN2FIX Labelling effort under the *Decision Tree* algorithm



(a) Proportion of generated (left) / failing tests that are labelled (right)



(b) Probability to generate (left) / label a failing test (right)

Fig. 6: Variations of labelling effort under the maximum number of queries 5,10,20,30,40 and 50

that the human receives more failing tests as the number of queries increases in LEARN2FIX.

5.3 RQ.3. Oracle Representation

Table 2 and Table 3 show the oracle quality and labelling effort of LEARN2FIX under the classification algorithms in Section 3.2, respectively. Not all classifier representations are capable of accurately modelling the failure condition of a bug. Thus, it is necessary to empirically evaluate the most appropriate representation for this task.

> LEARN2FIX trains better automatic oracles through *interpolation-based* classification algorithms than *approximation-based* classification algorithms. The median recall-failing is above 75% in these algorithms (Table 2). *Decision Tree* and *AdaBoost* algorithms generate the best automatic oracles with LEARN2FIX.

Interpolation-based approaches work better with LEARN2FIX than approximation-based approaches. Both *Decision Tree* and *AdaBoost* are better than *INCAL* in terms of oracle quality. The median precision-failing of *INCAL* is significantly lower (the difference is greater than 10%) than that of both algorithms (Table 2). According to the two-sided *Wilcoxon test*, the differences in the metrics between *Decision Tree* and *AdaBoost* are statistically insignificant ($p > 0.05$).

In approximation-based approaches, only *Naïve Bayes* produces automatic oracles that identify test failures with significant accuracy ($> 60\%$) in most subjects. Even though *SVM*, *MLP(20)* and *MLP(20,5)* show more than 70% overall median accuracy, their median recall-failing is below 50% (Table 2).

> Under all the classification algorithms, LEARN2FIX sends less than half ($< 50\%$) of the generated tests for labelling in most subjects (Table 3). The interpolation-based approaches *Decision Tree* and *AdaBoost* show around 60% median probability to label a failing test, which is approximately three times greater than finding a failing test by random labelling.

In the interpolation-based classification algorithms, more than 70% of the generated failing tests are sent for labelling in most subjects (Table 3). LEARN2FIX shows the highest median probability values for labelling a failing test case. According to the two-sided *Wilcoxon test*, the differences between *AdaBoost* and *Decision Tree* in the probability of labelling a failing test are insignificant ($p > 0.05$).

Even though the approximation-based approaches send only less than half of the generated tests, not the majority of the generated failing tests is sent for labelling. In *SVM* and *Naïve Bayes*, at least 10% of the generated failing tests is not sent for labelling in most subjects. This implies that the DECIDE2LABEL-algorithm works better with interpolation-based approaches than with approximation-based ones. The DECIDE2LABEL-algorithm achieves its intended objective, i.e., maximising the probability of sending failing tests, with interpolation-based approaches well.

When considering both labelling effort and oracle quality, the interpolation-based classification algorithms more effectively use the available query budget to improve the oracle quality than the approximation-based approaches. We believe that interpolation-based classification algorithms can incrementally model the condition under which a semantic bug is exposed more accurately than approximation-based ones. Hence, the DECIDE2LABEL algorithm is able to send more of the generated failing tests and increase the probability of labelling a failing test in oracle learning. This is helpful in dealing with the class imbalance problem. All these facts lead to high-quality automatic test oracles with the interpolation-based classification.

Among the approximation-based approaches, the *Naïve Bayes* algorithm differently behaves from the other approaches. It uses fewer failing tests than most classification algorithms (Table 3); however, the recall-failing and precision-failing of the automatic test oracles are above 60% for the median subject. This implies that *Naïve Bayes* can learn automatic test oracles at considerable accuracy using fewer training data. However, the DECIDE2LABEL algorithm fails to improve the test oracles obtaining more failing tests in *Naïve Bayes*.

5.4 RQ.4. Patch Quality

We investigate the quality of the patches generated using the test inputs that LEARN2FIX sent for labelling. The statement coverage of the manual and auto-generated test suites is

Classification Algorithm	Overall Accuracy (%)		Recall Failing (%)		Precision Failing (%)		Recall Passing (%)		Precision Passing (%)	
	Mean	Median	Mean	Median	Mean	Median	Mean	Median	Mean	Median
Interpolation-based										
INCAL	80.74	81.68	71.68	76.65	59.30	58.56	79.29	84.93	82.80	92.09
Decision Tree	85.01	88.95	**72.44**	**79.69**	71.07	75.75	84.93	93.53	84.57	**94.21**
AdaBoost	**85.38**	**89.34**	70.64	77.05	**74.02**	**79.01**	**85.87**	95.31	**85.25**	94.10
Approximation-based										
SVM	77.70	82.46	39.51	31.25	58.79	58.24	77.51	**97.27**	80.41	87.27
Naïve Bayes	79.25	83.04	63.82	65.63	66.12	68	80.34	92.12	81.46	89.39
MLP (20)	72.43	72.35	48.15	47.77	39.67	33.33	70.67	73.68	79.09	86.10
MLP (20,5)	72.03	72.07	47.68	46.88	39.08	31.96	70.81	74.53	77.56	85.43

TABLE 2: Mean and median values of the oracle quality of LEARN2FIX under different classification algorithms

Classification Algorithm	Percentage of generated tests that are labelled		Percentage of Failing tests that are labelled		Probability to generate a failing test (%)		Probability to label a failing test (%)	
	Mean	Median	Mean	Median	Mean	Median	Mean	Median
Interpolation-based								
INCAL	**36.92**	**31.09**	64.13	70.26	30.20	21.55	48.08	47.92
Decision Tree	31.45	22.78	**64.77**	**76.71**	**30.65**	22.70	**59.68**	**62.11**
AdaBoost	29.64	18.1	62.42	71.70	30.60	**22.97**	59.13	61.63
Approximation-based								
SVM	17.33	0.38	22.26	1.19	30.26	21.63	45.63	50.15
Naïve Bayes	17. 53	1.51	28.80	7.6	29.58	20.72	53.49	54.40
MLP(20)	33.90	23.71	45.21	42.08	30.36	21.72	39.97	34.53
MLP(20,5)	28.89	10.70	38.95	22.37	30.36	21.14	40.07	34.84

TABLE 3: Mean and median values of the labelling effort of LEARN2FIX under different classification algorithm

computed as well. Related to each APR technique, we count the number of repairable subjects in terms of defect categories under all the repair test suites. Table 4 summarises the results of program repair experiments under all the classification algorithms. Figure 7 shows the statement coverage of the manual and LEARN2FIX auto-generated repair test suites. Figure 8 shows the composition of the repair test suites.

For both test-driven APR approaches, the patches produced using auto-generated test suites outperform the patches produced using the manual test suites in terms of the validation score of the generated patches. For both APR approaches, *all* (100%) test cases in the held-out test suite pass on the majority of subjects when interpolation-based classification algorithms are used. Both types of test suites can repair less than 30% of the selected subjects. While with manual test suites the APR tools can repair more subjects than with the auto-generated test suites, the quality of the generated patches is better for auto-generated test suites.

Number of failing tests. Each manual repair test suite given by Codeflaws contains a single failing input exposing the bug (Figure 8). This kind of repair test suite could lead to producing an overfitting or incorrect patches [18]. In contrast, the auto-generated repair test suite given by LEARN2FIX contains more than one failing test in most subjects, as DECIDE2LABEL algorithm prioritizes the labelling of failing tests. Thus, it can exercise the faulty behaviours of the bug more precisely than the manual repair test suite. For this reason, the APR tools can produce more accurate and non-overfitting patches with the auto-generated repair test suites under most classification algorithms. *Interpolation-based oracle*. The auto-generated repair test suites with the interpolation-based classification algorithms show 100% validation score for the median subject (Table 4). It implies that an auto-generated repair test suite under these algorithms contains enough failing tests to indicate the bug and enough passing tests to indicate the behaviour that should not be changed. In **RQ.3.**, we observed that interpolation-based classification algorithms can train highly accurate automatic oracles with LEARN2FIX (Table 2). Therefore, most generated failing tests and the passing tests in the vicinity of those are sent for labelling. As a result, repair test suites leading to high-quality program repair are generated.

Approximation-based oracle. The auto-generated repair test suites with *SVM* do not outperform the manual test suites in terms of the validation score in any APR tool. LEARN2FIX sends very few generated tests for labelling with *SVM* (Table 3). Figure 8 shows that the number of passing tests in these repair test suites is lower than the other auto-generated repair test suites and manual test suites (SVM box). On average, there are four (4) passing tests in an *SVM* repair test suite, which is the lowest compared to the others. Consequently, such a repair test suite cannot completely exercise the behaviour that should not be changed. The reductions in statement coverage (Figure 7) in these repair test suites also imply this fact. LEARN2FIX generates smaller repair test suites with lower statement coverage with *Naïve Bayes* as well. Nevertheless, these repair test suites perform better than the repair test suites generated with *SVM*.

Search-based versus constraint-based APR. The results in

Test Suite	GenProg				Angelix			
	Repairability (%)		Validation Score(%)		Repairability (%)		Validation Score(%)	
	Mean	Median	Mean	Median	Mean	Median	Mean	Median
Auto-generated Interpolation-based								
INCAL	17.14	17.15	90.35	**100**	15.66	15.57	90.65	**100**
Decision Tree	16.48	16.48	**90.45**	**100**	16.49	16.45	**90.66**	**100**
AdaBoost	16.47	16.54	89.71	**100**	16.19	16.29	90.44	**100**
Auto-generated Approximation-based								
SVM	**24.15**	24.13	81.77	94.52	22.71	22.70	82.64	91.67
Naïve Bayes	21.06	21.08	84.64	97.3	20.97	20.89	84.21	94.44
MLP (20)	18.64	18.51	86.75	**100**	18.49	18.50	88.51	97.14
MLP (20,5)	18.91	18.94	86.63	**100**	18.41	18.60	88.13	95.83
Manual								
Manual	23.52	**24.50**	85.14	97.56	**25.53**	**25.50**	83.94	91.67

TABLE 4: Mean and median values of the repairability and validation score under GenProg and Angelix

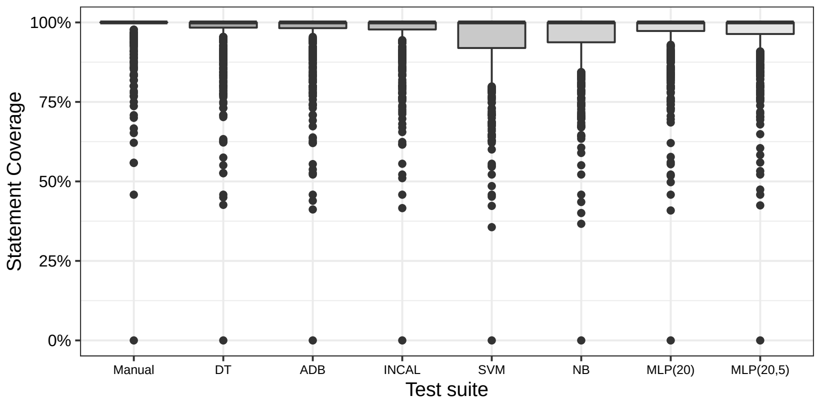


Fig. 7: Statement coverage of Codeflaws manual repair test suites and LEARN2FIX auto-generated repair test suites under different classification algorithms

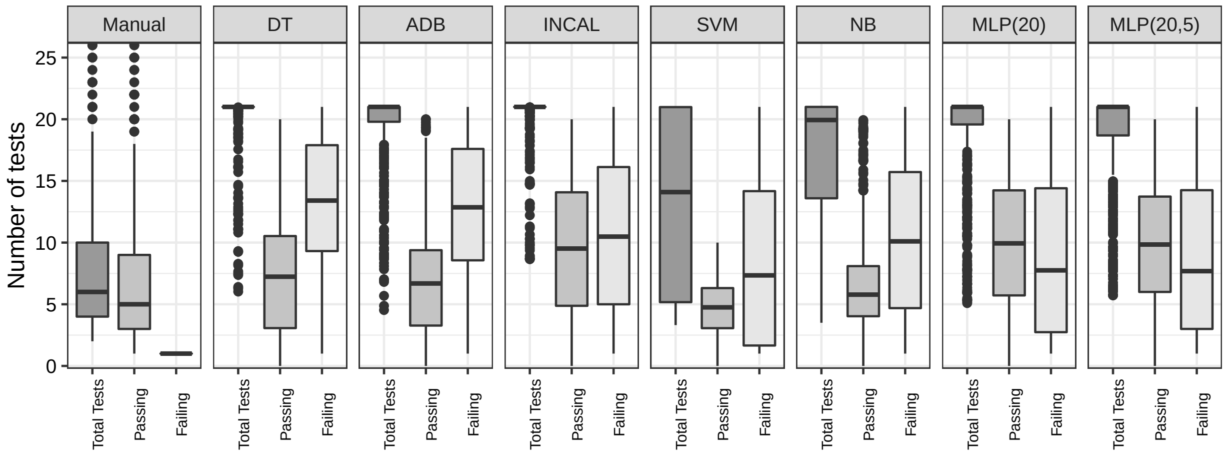
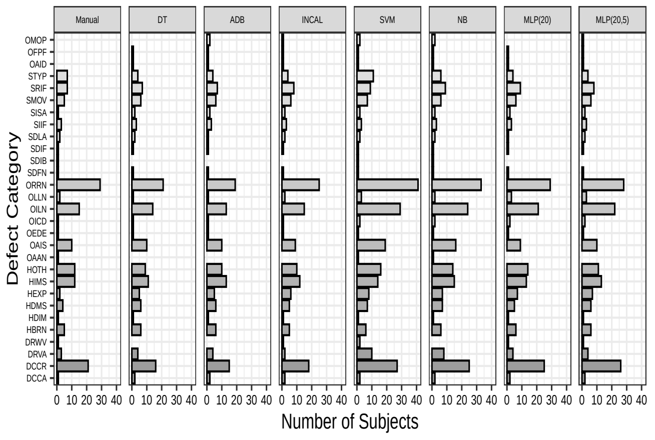


Fig. 8: Composition of the repair test suites (Manual and Auto-generated)


(a) GenProg

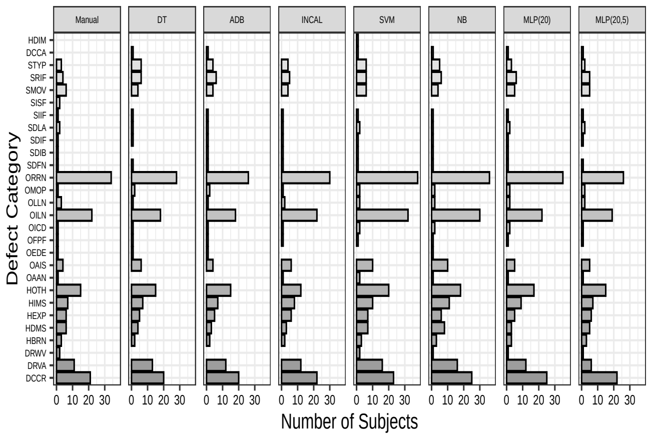

(b) Angelix

Fig. 9: Number of repairable subjects in GenProg and Angelix by defect categories under manual and auto-generated test suites

Table 4 indicate that *GenProg* works more successfully with the auto-generated repair test suites than *Angelix*. Except *SVM* and *Naïve Bayes*, the classification algorithms show 100% validation score for the median subject in *GenProg*. Only the interpolation-based approaches achieve this much accuracy in *Angelix*. Using the manual repair test suites, *GenProg* produces more accurate patches than *Angelix*.

Code coverage of auto-generated test suites. According to Figure 7, the auto-generated test suites cover all (100%) code in the median subject. This is same in the manual test suites as well. These results imply that LEARN2FIX's semi-automatic approach can generate a repair test suite that completely covers the code in most programs. In a few subjects, we observed that the manual repair test suite shows higher statement coverage than the auto-generated test suites. The reason is that the DECIDE2LABEL algorithm sends failing tests for labelling with priority. Consequently, passing test cases that cover certain code segments might not be sent for labelling. Another observation in Figure 7 is that the auto-generated repair test suites with *SVM* and *Naïve Bayes* report lower code coverage than the other auto-generated test suites (SVM and NB boxes). As explained before, we believe that the incapability of these algorithms to send more test inputs for labelling in oracle learning is the reason for this outcome. Even though the manual repair test suites contain only one failing test, those have been created to achieve 100% statement coverage. However, the single failing test can be insufficient to exercise all the faulty behaviours of the bug, which leads the APR tools to produce lower-quality patches.

Defect categories. Figure 9 shows the distributions of repairable subjects by the defect categories (Table 1) in both APR tools. Most repairable subjects by both manual and auto-generated test repair test suites belong to **ORRN**, i.e., replace relational operator. In both APR techniques, we observe lower repairability in the auto-generated repair test suites than in the manual ones (Table 4). The technical issues associated with the APR techniques are the key reason for this situation. When a repair test suite contains many failing test cases, the APR technique might not be able to produce a patch that passes all the failing tests within the allocated time. However, less accurate patches, similar to the ones generated with the manual test suites, do not fix bugs completely and can create new bugs in programs. Therefore, we conclude that the auto-generated test suites under the interpolation-based approaches are more suitable for automated program repair than the manual test suites.

LEARN2FIX auto-generated test suites and the manual test suites show 100% statement coverage in most subjects. In both APR tools, most repairable subjects under the auto-generated and manual repair test suites belong to the **ORRN**, i.e., replace relational operator, category.

5.5 RQ.5.: Impact of Noisy Labels

We investigate the impact of mislabelling on the oracle quality and labelling effort under the best-performing classifier (decision tree) and a budget of 20 queries. The human can make mistakes in labelling tests. As LEARN2FIX uses human-labelled tests to train automatic oracles and to repair programs, it is important to analyze the impact of incorrectly labelled tests. We allow between 0% and 20% of labelling queries to be incorrect (noise levels). Figure 10 shows the distributions of the overall accuracy, failing-recall/precision and passing-recall/precision.

The oracle quality decreases under incorrectly labelled test cases. The highest decreases can be seen in precision for failing tests (failing-precision box).

According to Figure 10, as expected incorrectly labelled test cases negatively affect the oracle quality. Furthermore, the automatic oracle's ($\mathcal{O}$) ability to identify both passing and failing tests reduces when the test cases are incorrectly labelled in LEARN2FIX active oracle learning (Algorithm 1). Figure 10 further implies that the precision for failing inputs is significantly affected by incorrectly labelled test cases. The

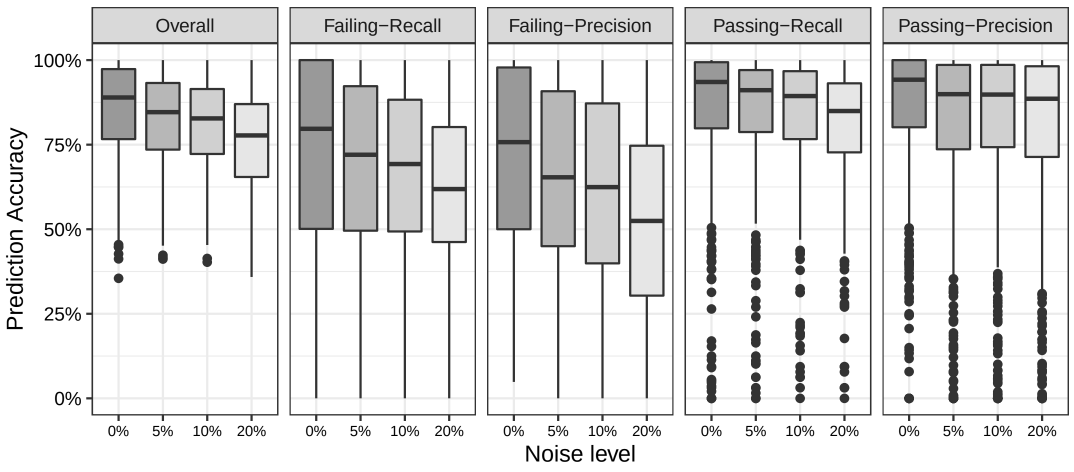


Fig. 10: Variations of oracle quality under the noise levels 5%, 10% and 20%

median failing-precision drops by 10% when 5% noise is present (failing-precision box).

> There is a significant drop in effectiveness of LEARN2FIX to accurately identify failing test inputs to be sent for labelling. The number of failing test cases sent for labelling reduces under incorrectly labelled test cases. The probability of labelling a failing test decreases, as well.

Impact on oracle quality. Figure 11 shows how the labelling effort varies under the previous noise levels. Maximizing the labelling of failing test cases is an objective of LEARN2FIX active oracle learning. According to the results in Figure 11, incorrectly labelled test cases prevent achieving this objective. As test cases are incorrectly labelled, LEARN2FIX's ability to select failing test cases for labelling decreases. Figure 11a-right shows this fact. When the noise level is 20%, LEARN2FIX sends less than 50% of the generated failing test cases for labelling in most subjects. As LEARN2FIX misses failing tests, the test cases sent for labelling are not frequently failing tests. Thus, the probability of labelling a failing test case reduces as in Figure 11b-right. In the noise levels 10% and 20%, the median probability of labelling a failing test is below 50%.

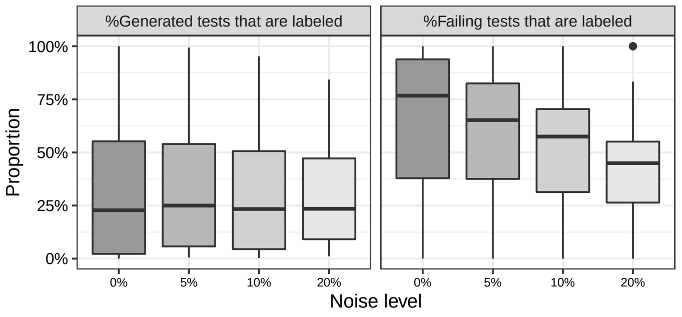
Interpretation. LEARN2FIX starts the oracle learning from a single failing test case and incrementally expands the training test suite (T). The label given by the user in one step affects the subsequent steps of the learning process. In the beginning, the training test suites do not contain many test cases. Therefore, the oracle accuracy is significantly affected when the user incorrectly labels a test case in the initial stages. When the automatic oracle is inaccurate, Algorithm 2 cannot correctly select the failing test cases in the subsequent steps for human labelling. The decreasing percentages of human-labelled failing tests in Figure 11a-right demonstrate this fact. In addition, when failing tests are labelled as passing, the chance to explore more failing tests is reduced. When passing test cases are labelled as failing, those are used to generate new test cases by fuzzing(Algorithm 1-Line 5 and 10). Finding more failing test cases by mutating passing test cases is difficult. All these facts reduce LEARN2FIX's ability to explore failing test cases. The lack of failing tests in T reduces the automatic oracle's ability to correctly identify failing tests, resulting in a drop in the oracle quality.

> When there are incorrectly labelled test cases, the repairability of LEARN2FIX auto-generated test suites decreases in both GenProg and Angelix. Moreover, the validation score of the patches decreases as well.

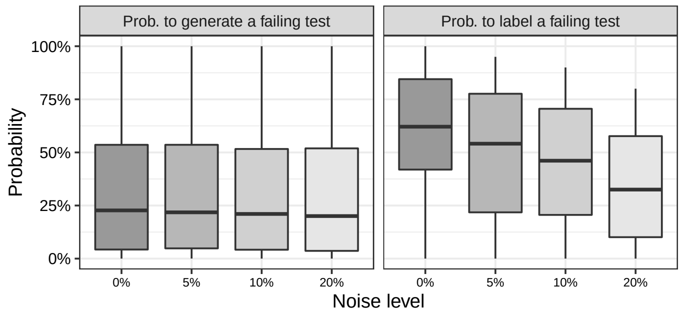
Impact on repairability. The test suites generated by LEARN2FIX under the above noise levels were used to repair the programs using GenProg and Angelix. Figures 12a and 13a indicate that the ability of LEARN2FIX auto-generated test suites to repair buggy programs is reduced when there are incorrectly labelled test cases. The drop in repairability in GenProg is more significant than in Angelix under the noise level. In GenProg, from 0% to 5% noise, the median repairability drops from 16% to 1% (Figure 12a). One reason for the lower repairablity in GenProg is associated with its fault localization technique. GenProg uses Equation 14 to calculate the suspiciousness of a statement [1].

$$suspG(s) = \begin{cases} 0, \ \mathit{failed}(s) = 0 \\ 1.0, \ \mathit{passed}(s) = 0 \wedge \mathit{failed}(s) = 1 \\ 0.1, \ \mathit{otherwise} \end{cases} \tag{14}$$

Interpretation. According to Equation 14, the suspiciousness of a statement is highest if it is only executed by failing test cases ($1.0, \ \mathit{passed}(s) = 0 \wedge \mathit{failed}(s) = 1$). A single passing test case is enough for reducing the score of a statement from 1.0 to 0.1. If a failing test case is incorrectly labelled as passing, the faulty statements executed by that test case receives 0 or 0.1. Consequently, GenProg cannot correctly identify the actual faulty statements. As a result, GenProg tries to change unnecessary statements and produces no repair. The same problem can be observed under passing test cases incorrectly labelled as failing as well. In addition, we observed in the experiments that the search space gets larger with incorrectly labelled test cases, and, therefore, GenProg cannot produce a repair within the allocated time.

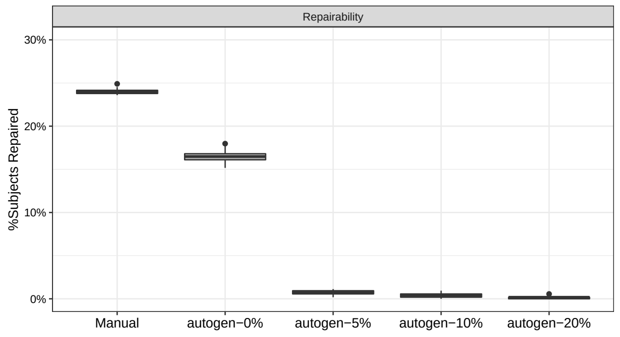


(a) Proportion of generated (left) / failing tests that are labelled (right)

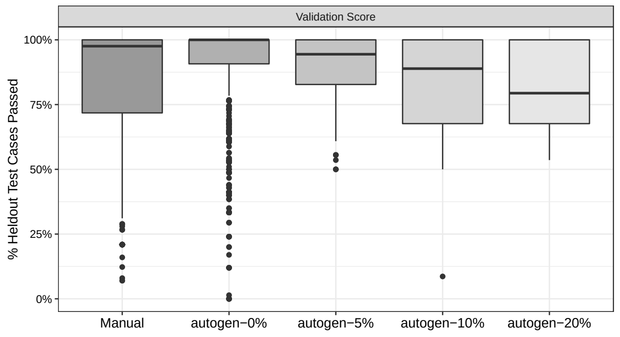


(b) Probability to generate (left) / label a failing test (right)

Fig. 11: Variations of labelling effort under the noise levels 5%, 10% and 20%

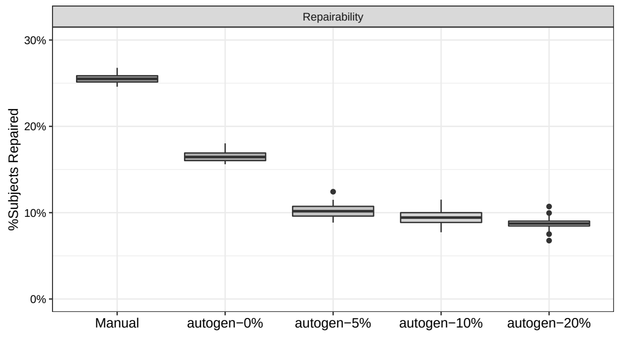


(a) Repairability

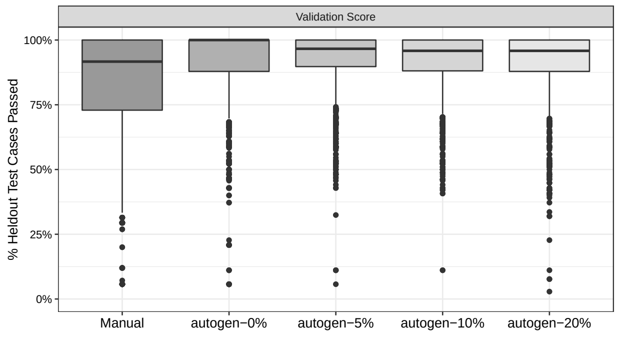


(b) Validation Score

Fig. 12: Variations of repairability and validation score in GenProg under the noise levels 5%, 10% and 20%.



(a) Repairability



(b) Validation Score

Fig. 13: Variations of repairability and validation score in Angelix under the noise levels 5%, 10% and 20%.

All these facts lead to the significantly lower repairability of GenProg under incorrectly labelled test cases. In contrast to GenProg, Angelix uses *Jaccard formula* (Equation 15) for the fault localization [15].

$$suspJ(s) = \frac{failed(s)}{execute(s) + (totalFailed - failed(s))} \tag{15}$$

Jaccard formula is not sensitive to incorrectly labelled test cases as Equation 14, i.e., an incorrectly labelled test case cannot significantly reduce the suspiciousness score of a statement when there are enough correctly labelled test cases. Therefore, the drops in repairability in Angelix under different noise levels are not as large as in GenProg.

Impact on patch quality. Figure 12b and Figure 13b indicate decreases in the median validation scores under the auto-generated test suites with incorrectly labelled test cases in both APR tools. According to the one-sided *Wilcoxon-test*, the observed decrease in the validation scores in GenProg are statistically significant ($p < 0.05$). In Angelix, statistically significant decreases can be seen from the 10% noise level. Compared to the manual repair test suites, the validation scores of the auto-generated repair test suites are higher in Angelix, even though incorrectly labelled test cases exist. In GenProg, all the median validation scores of the auto-generated repair test suites with incorrectly labelled tests are below the median of the manual test suites.

Interpretation. The higher sensitivity to incorrectly labelled tests in GenProg's fault localization technique leads to reducing the validation score of the generated patches. GenProg follows a generate-and-validate approach that highly depends on the fault localization information [12]. When the fault localization is faulty, the generated patch becomes less accurate. These drawbacks cannot be seen in Angelix. As described before, the Angelix's fault localization is less sensitive compared to GenProg fault localization. Also, after finding the faulty program statements, Angelix uses a constraint solving approach to generate the patch [15]. This approach is not significantly affected by a few incorrectly labelled tests when there are enough correctly labelled tests. For this reason, the validation scores of the patches are not reduced as in GenProg.

Incorrectly labelled test cases negatively affects the oracle quality, labelling effort and patch quality in automated program repair. Test-driven APR techniques assume that the repair test contains correctly labelled test cases. Thus, incorrectly labelled tests significantly affect the patch quality of the auto-generated test suites. The classification algorithms cannot produce accurate oracles with incorrectly labelled test cases. As a result, LEARN2FIX becomes unable to send failing tests more frequently for human labelling. The lack of failing tests also reduces the patch quality.

6 PILOT USER STUDY

We conducted a pilot user study with six (6) participants to assess the usability of LEARN2FIX in an actual human-in-the-loop environment. In this study, we especially focused on the human effort involved in deciding the label of a test case and providing the expected, correct output of a failing test case.

Study Design. We used the program in the motivating example (Listing 1) as the buggy subject ($\mathcal{P}$). The *Decision Tree* (DT) algorithm was used as the classification algorithm of LEARN2FIX based on the results in Section 5.3. We implemented a user interface for the participants to interact with LEARN2FIX.

At the beginning of the study, each participant was given a brief introduction to the task, including a demonstration of the user interface. Also, they were informed that the intended functionality of the buggy subject (Listing 1) is to classify triangles based on the lengths of their sides. However, the source code of the program was not revealed to them. For each participant, we allocated **10 minutes** to interact with LEARN2FIX and set the maximum number of labelling queries to 20 ($L = 20$).

As described in Algorithm 1, a participant was presented with a series of labelling queries to learn an automatic oracle. A labelling query shows the values of the three inputs and the corresponding output given by the buggy subject. Then, the participant has to answer the question "Did the program return the correct output ?". If the answer is "No", the participant is asked to provide the correct output. Otherwise, the next labelling query is presented to the participant. Each participant was allowed to interact with LEARN2FIX until the timeout was reached, or the allocated query budget is exhausted. At the end of oracle learning, the participant was allowed to provide any feedback on our approach.

While interacting with LEARN2FIX, we measured the time that the participant spent answering each labelling query. In the test cases labelled as failing, the total time for deciding the label and providing the correct output was measured. At the end of oracle learning, we calculated the oracle quality and labelling effort. To measure the oracle quality, we used a manually constructed and labelled test suite.

Selection of Participants. We followed the *Belmont principle* [38] of respect people. We introduced ourselves, explained that they are being asked to participate in our pilot study, described the study procedures and how their anonymized data is used, explained that the participation is voluntary, and provided our contact information for questions and concerns about our research. The six people who agreed to participate in our pilot user study had sufficient mathematical knowledge to understand the triangle classification problem. This kind of person is eligible for this study, as we assume that the human interacting with LEARN2FIX knows the expected behaviour of the program under test.

	Mean (%)	Median (%)
Oracle Quality		
Overall Accuracy	90.91	90.91
Recall-Failing	100	100
Precision-Failing	75	75
Recall-Passing	87.50	87.50
Precision-Passing	100	100
Labelling Effort		
Proportion of generated tests that are labelled	1.7	0.2
Proportion of labelled failing tests from generated	94.71	100
Probability to generate a failing test	0.18	0.12
Probability to label a failing test	58.39	66.06
	Min (s)	**Max (s)**
Response time (Seconds)		
Answering a labelling query (Including providing the expected output of a failing test case)	2	40

TABLE 5: Summary of the results in the pilot user study

Results. Table 5 summarizes the results of our pilot user study. A user took **2 - 40** seconds to answer a labelling query, including the time for providing the expected outputs of a failing test case. Only one participant incorrectly labelled *two passing tests* as *failing*. All the other participants made no mistakes in answering the queries. Only two participants received 20 queries to answer, and all the others received less than that in the allocated time.

Similar to Section 5.1, LEARN2FIX produces high-quality automatic oracles for the program in Listing 1. There is a rare chance of generating a failing test for this program (i.e., the probability of generating a failing test is $< 1\%$). Also, very few tests from the generated tests are sent for human labelling. Nevertheless, LEARN2FIX can select most generated failing tests and let the participant receive those more frequently. This outcome is similar to the results obtained in Section 5.2.

User Feedback on Labelling Queries. All the participants claimed that the queries presented were easy to answer. Also, they said that the time to generate the next query after answering the current query was significant in some situations. Most participants received the passing tests in the first few queries and the failing tests in the later queries. The reason for these incidents is that the automatic oracle becomes accurate as more test cases are received; therefore, the DECIDE2LABEL algorithm accurately selects the failing tests while excluding most generated passing tests. As explained in Section 3.1, the failing tests of Listing 1 are rarely generated. The results in Table 5 also confirm this fact. Due to the tendency of LEARN2FIX to select failing tests for human labelling, the time to generate the queries can be longer.

Additional User Feedback. We received some additional important feedback regarding LEARN2FIX from the participants in this pilot user study. Firstly, they mentioned that this is a practical approach, as the user (i.e., person who answering) only needs to know the expected behaviour of the system under test. Experience in programming is optional for working with LEARN2FIX. Secondly, the participants got familiar with the pattern of the failing tests as answering the queries. As a result, some participants answered the last few queries quicker than the earlier ones. This is also an advantageous property in LEARN2FIX. In addition, a few participants commented that the chance of occurring errors in labelling tests is minimal in real scenarios, as a person (e.g. developer) trying to fix a bug usually studies the expected behaviour of the given program.

To summarize, LEARN2FIX successfully works with the actual human participants for the selected buggy subject. The user response time for labelling queries and user feedback indicate that a human participant can easily answer the labelling queries. This is a reasonable outcome, as we selected participants who know the expected behaviour of the subject buggy program. Similar outcomes could be expected for other programs if it is possible to find a user who correctly knows the expected behaviour of the program under test (PUT) to work with LEARN2FIX. However, the response time for the labelling queries could vary depending on the complexity of the PUT and the skill of the human participant.

7 Threats to Validity

Similar to the other empirical studies, there are various threats to the validity of our results and conclusions. The first concern is the *external validity*, i.e., to what extent our findings can be generalized to and across other subjects and tools. Our results may not hold for other subjects. The classification algorithms in Section 3.2 work only for programs taking numeric data. Thus, our program subjects were required to take numeric input values and return numeric output values. Nevertheless, we selected a large number of real, arithmetically complex faulty programs under diverse defect categories (i.e., 552 programs under 34 defect categories). To answer **RQ.4.**, we used GenProg [12] and Angelix [15] as the automated program repair tools. GenProg is a heuristic / generate-and-validate repair technique, whereas Angelix is a constraint-based repair technique. These two techniques are state-of-the-art and have been shown to repair large open-source programs cost effectively. The results of the pilot user study (Section 6) demonstrate that LEARN2FIX would work with real human participants. However, we used only the program in Listing 1 in the study, as conducting this kind of experiment for a large group of programs is impractical. The key reason is that finding a group of participants who are familiar with the expected behaviours of a large set of programs is difficult. The expected behaviour of Listing 1, i.e., classifying triangles based on side lengths, is a famous and simple concept in geometry, hence easier to find participants for the study. The challenges faced by the human might be different in labelling test cases in other programs.

The second concern is *internal validity*, i.e., to what extent our study minimizes the systematic error. For each subject, we repeat each experiment 30 times and report the average values of the metrics. This approach helps to mitigate spurious observations due to the randomness of the mutational fuzzer and classification algorithms. Also, it helps to gain statistical power for the results. Similar to other implementations of other techniques, our tool may not faithfully implement LEARN2FIX as presented in Algorithm 1 and 2. However, to facilitate scrutiny and reproducibility, we make the source code and all data available.

The third concern is *construct validity*, i.e., to what extent a test measures what it claims to be measuring. To reduce this threat, we discuss at least two measures for each of the three independent variables: oracle quality, human effort and patch quality.

8 Related Work

Test oracle automation is an important topic in automated software engineering. Several surveys on the oracle problem, such as the works of Earl et al. [39] and Pezzè et al. [40], emphasise the importance of having automated test oracles in test automation. However, this topic has received significantly less research attention than the other areas in automated software engineering [39]. Also, the work of Briand [23] suggests that test oracle automation is probably one of the most difficult problems in software testing. All these facts indicate that significant improvements are required in the area of test oracle automation.

The survey paper of of Earl et al. [39] categorises test oracles into three groups as *implicit oracles*, *specified oracles* and *derived oracles*. *Implicit oracles* use general implicit knowledge of incorrect program behaviours, such as program crashes and timeouts. Also, implicit oracles can be *injected*. As an example, ASAN [41] induces a crash for an input exposing a memory safety error. However, this category of oracles cannot be used for semantic bugs or functional bugs. *Specified* *oracles*, i.e., test oracles based formal specifications, are ideal for semantic bugs. However, developing these oracles is impractical in most scenarios due to the difficulty of finding a formal specification of a program [39]. For this reason, *derived oracles*, i.e., oracles derived from sources other than formal specifications, are most suitable for semantic bugs. Based on these facts, we focused on a method to derive test oracles based on test cases. *Explicit oracles* are applied to detect functional and other bugs and should be manually added. Typically, developers introduce explicit oracles to programs as assertions [42]. These assertions should be added proactively. However, our objective is to develop oracles retroactively, i.e., the oracle should identify new test cases exposing a known failure.

The works of Jin et al. [26], Vanmali et al. [43] and Shahamiri et al. [44], [45] are some *supervised machine learning* [46] based oracle learning approaches. The automatic oracles given by these methods are *black-box*, i.e., only the program inputs and outputs are used to determine test failures. All these works use *artificial neural networks* to learn the relationship between (i.e., the function) the program inputs and outputs. The learned function can be explicitly represented as program assertions or likely invariants [47]. Given an input, the neural network model or the learned function predicts the *expected output*. The predicted output is compared with the output produced by the program under test for the same input. If the two outputs are similar, the test is predicted as *passing*; otherwise, it is *failing*. To learn an accurate model for this task, these works ([26], [43], [44], [45]) require a large training test suite. When the human is the only oracle, finding such a test suite is a challenging task.

In contrast to these works, LEARN2FIX learns the condition under which the bug is exposed and thus produces bug oracles. We believe that this is not a complicated task as learning the relationship between the program inputs and outputs. Also, it can be done more efficiently. For instance, in our motivating example (Section 2), LEARN2FIX does *not* learn how to classify a triangle. Instead, it learns which triangles the program in Listing 1 incorrectly classifies. The work of Braga et al. [22] uses *AdaBoost* to produce bug oracles based on user actions. However, this approach is only applicable to web applications. In contrast, LEARN2FIX can be applied to a wide range of programs taking numeric inputs.

The active learning method used in LEARN2FIX was inspired by the work of Holub et al.[9]. The key objective of this work is to reduce the human labelling effort during image classification by sequentially presenting unlabelled images that are informative when labelled to the human (or oracle). In each iteration, Holub's method selects the *most informative unlabelled point* (MIUP), i.e., the image that the classifier is most uncertain about its label, based on the current status of the classifier. The work of Joshi et al. [48] is similar to this work.

LEARN2FIX uses the concept of *look-ahead probability estimation* from Holub's work [9] to select test cases with higher failure likelihood for human labelling. This is one way that LEARN2FIX addresses the class imbalance problem. Moreover, we extended Holub's *pool-based* approach, where the number of data points is fixed, into *stream-based* approach, where data points are continuously generated and decided upon. Our insight is that a reliable probability estimate for a point's label can be derived with a fixed-size random classifier committee.

The idea of generating more failing tests by mutating a single failing test has been applied in AFL fuzzer [20] and its recent developments [49], [50], [51]. The main focus of these works is to explore the bugs leading to program crashes. Generating new test cases helps to isolate faults (BUGEX [52]) and improve auto-generated patches [53]. Given only a stack trace, there exist techniques to generate crashing inputs [54] [55]. Unlike in our work, an automated oracle, probably program crashes, is already assumed in all these works.

LEARN2FIX interacts with a human oracle in the learning process, and reducing the human effort is an important consideration in our work. Several works have been proposed to reduce the effort of human oracles in *qualitative* and *quantitative* aspects [39]. The quantitative approaches focus on reducing test suite and test case size. The works of Harman et al. [56], Ferrer et al. [57] and Taylor et al. [58] focus on exercising *all the different behaviours* of the program under test with *fewer test cases*. The qualitative approaches focus on improving the comprehension of the tasks performed by the human as a test oracle. For example, the work of Afshan et al. [59] incorporates a natural language model into the test generation process to *improve the human readability* of the generated test cases. McMinn et al. [60] propose a method to facilitate *domain-aware* test generation by incorporating knowledge from programmers, source code and documentation into automatic test generation. This method can generate more human readable test cases as well.

In addition, Staats et al. [61] propose a technique to select *oracle data*, i.e., the subset of internal variables that should be monitored during testing. Focusing on these variables reduces the effort of human oracles. Distributing test cases among different users is another strategy to reduce the human effort. Pastore et al. [62] suggest an approach to present test cases to a crowd for labelling. However, none of these approaches addresses the problem of developing an automatic oracle by systematically labelling generated test cases. Nevertheless, the *qualitative* approaches could be useful to improve the human readability of our work.

GRAMMAR2FIX [63] is closely related to our work. Given a failing input of the semantic bug, GRAMMAR2FIX generates an automatic oracle and a repair test suites for the bug. The automatic oracle is given as a regular grammar that describes the pattern of all the failing inputs of the bug. Unlike LEARN2FIX, GRAMMAR2FIX cannot be applied under a limited number of human queries. The work of Bowring et al. [64] is another active oracle learning approach similar to our approach. In contrast to LEARN2FIX, this method uses some *white-box* information such as *event transitions*. To reach significant prediction accuracy, this approach requires more than 100 human labelled executions. In contrast, LEARN2FIX achieves high oracle quality with significantly fewer human queries (20), even without knowing the source code of the program under test.

Automated program repair (APR) is an emerging research area that focuses on reducing the cost of manual debugging while improving software quality [1][2]. In *test-driven* automated program repair, the quality of the patch depends on the quality of the repair test suite. Low-quality repair test suites lead to repair overfitting [1][18]. Yu et al. [3], Yang et al. [4] and Xiong et al. [5] present several methods to generate high-quality repair test suites for test-driven APR, avoiding repair-overfitting. Unlike LEARN2FIX, all these methods require an initial repair test suite (containing both passing and failing test cases). The given repair test suite is systematically augmented in a manner improving the quality of the patch. *UnsatGuided* by Yu et al. focuses on constrained-based repair techniques [3], while Yang's method [4] focuses on heuristic repair. Xiong's method [5] can be applied to both categories of repair techniques, similar to LEARN2FIX. However, Xiong's method needs a patch as an input in addition to a repair test suites. In contrast to all these methods, LEARN2FIX can be applied to all types of test-driven repair techniques, given a single failing input of a bug.

Evaluating the correctness of patches is an important task in the researches related to APR. Manual inspection (e.g. [34][65]) and using a validation test suite independent from the repair test suite (e.g. [37][18]) are the two main methods for evaluating patch correctness. The study of Motwani et al. [36] suggests that using an independent repair validation test suite is more objective than manual inspection. The study of Le et al. [34] reveals that manual-inspection-based methods can be inaccurate due to their subjective nature. For this reason, Le's study emphasises that the results of manual-inspection-based patch evaluations should be publicly available. In contrast, the patch evaluations using independent repair validation suites can be reproduced fully automatically.

The study of Yi et al. [66] explores the correlation between traditional test suite metrics proposed for software testing (e.g. statement coverage, test suite size, mutation score etc.) and the reliability of generated repairs by APR. Its conclusion is that the traditional metrics are useful for APR to improve the reliability of repairs. For example, this study shows that *regression-causing repairs* [67] (i.e. pass all positive tests, but fail one of positive tests) can be mitigated by improving the statement coverage of repair test suites. Our work also focuses on improving the quality of program repair by improving the quality of repair test suites.

9 Discussion and Future Work

Given a program with a semantic bug and its single failing test, LEARN2FIX learns a bug oracle as a classifier, which is the automatic oracle. The learned automatic oracle ($\mathcal{O}$) expresses the condition under which the bug is exposed (i.e., *failure condition* of the semantic bug) LEARN2FIX improves the overall oracle quality by improving the classifier's ability to correctly identify failing tests, the minority class. For this purpose, LEARN2FIX maximises human labelling of failing tests in the learning process. The results of oracle quality (Section 5.1) and labelling effort (Section 5.2) suggest that LEARN2FIX's oracle learning strategy works for many real world semantic bugs in programs taking numeric inputs. The automatic oracles show more than 75% accuracy in identifying both passing and failing tests for most subjects. Manually exploring the failing tests of a semantic bug is a difficult task in programs taking numeric inputs. LEARN2FIX effectively addresses this issue by its DECIDE2LABEL-algorithm. With the DECIDE2LABEL-algorithm, the probability of finding a failing test is *three* times higher than with random labelling. Thus, the human would receive failing test cases frequently, even though those are rarely generated. Hence, exploring failing tests would be easier for the human through LEARN2FIX rather than doing it solely by mutational fuzzing.

The experiments in Section 5.3 reveal that LEARN2FIX works as intended with few classification algorithms. If the classification algorithm is capable enough to accurately infer the failure condition of a semantic bug, the DECIDE2LABEL algorithm sends more failing tests generated by mutational fuzzing for human-labelling. As failing tests are the minority class, it is helpful in improving the oracle quality while dealing with the *class imbalance problem* [8].

Our results suggest that interpolating binary classifiers produce better automatic oracles than approximating binary classifiers with LEARN2FIX. It implies that interpolating binary classifiers can better represent the failure condition of a semantic bug; i.e., highly accurate automatic test oracles can be produced by finding a model that exactly fits the training data (human-labelled passing and failing test). The interpolation-based approaches used in the experiments create classifiers as a set of constraints on a numeric domain. According to the results, such constraints can effectively represent the failure condition of a semantic bug. According to Section 5.3, LEARN2FIX shows the best performance with *Decision Tree* and *AdaBoost*. *AdaBoost* is an ensemble version of decision trees. Therefore, the decision tree representation is most suitable for developing automatic test oracles for semantic bugs.

In approximation-based approaches, we observed higher oracle quality in *Naïve Bayes* than in the other approaches. *Naïve Bayes* uses the *Bayesian probability* model [68] to develop classifiers. Our results indicate that the *Naïve Bayes* algorithm can accurately learn the failure condition of a semantic bug as a probability model with fewer training data. However, this algorithm does not outperform the interpolation-based approaches in terms of oracle quality. Also, it does not perform well in the program repair experiments.

As the maximum number of queries to the human increases, the automatic oracle's ability to distinguish between passing and failing tests increases as well (Figure 4). This is an intuitive outcome in machine learning, as the classification algorithm receives more data to learn an automatic oracle. In addition, the probability of labelling a failing test case also increases under the increasing query budget (Figure 6). It implies that LEARN2FIX always uses the query budget targeting failing tests and does not send passing tests more frequently, even though human queries are available.

The experimental results related to automated program repair (Section 5.4) suggest that LEARN2FIX produces high-quality repair test suites with the interpolation-based approaches in *GenProg* and *Angelix*. These approaches produce high-quality automatic test oracles for semantic bugs (Section 5.3). Therefore, these results imply that classification algorithms that produce high-quality test oracles generate high-quality repair test suites for both APR techniques.

Finding a repair test suite leading to high-quality fixes, i.e., non-overfitting and accurate patches, is a challenging task in automated program repair. This issue becomes more intense in semantic bugs, as only a human can answer about a test failure. Providing an effective answer to this issue, LEARN2FIX facilitates a *human-in-the-loop interactive program repair* environment. Even a person without experience in programming can contribute to the LEARN2FIX's program repair process.

In LEARN2FIX, we assume that the human always provides the accurate label of a test case. However, the human can make mistakes in deciding the label of a test case in practical scenarios. In Section 5.5, we explored the consequences of incorrectly labelled test cases in oracle learning and APR. The results suggest that LEARN2FIX cannot achieve its objectives when the human provides incorrect labels. Moreover, the repairability of the auto-generated repair test suites and the correctness of patches are reduced. The reason for all these issues is the incremental learning process in LEARN2FIX, i.e., if the human makes a mistake at one point, it affects the subsequent oracle learning steps. This is a drawback in LEARN2FIX.

According to the results in Figure 12 and Figure 13, a higher impact from incorrectly labelled tests can be seen in *GenProg* than in *Angelix*. GenProg's fault localization method is significantly misguided by incorrectly labelled test cases. The result is that GenProg produces no repair or less accurate repairs. Indeed, *Angelix*'s program repair methodology cannot be misguided by a few incorrectly labelled tests when the repair test suite contains sufficient correctly labelled tests (see Section 5.5). This is an advantage in Angelix over GenProg.

Test-driven APR techniques always assume that the repair test suite has correctly labelled test cases. Thus, generating no repair or incorrect repair can be expected when there are incorrectly labelled test cases in the repair test suite. The results of **RQ.5.** indicate this fact. Even the manual repair test suites would not achieve this much repairability and validation score if it contained incorrectly labelled tests. All these facts imply that repair test suites for test-driven APR should be prepared under scrutiny.

In future work, we will explore techniques to deal with incorrectly labelled test cases in oracle learning. The errors in labelling test cases can be avoided. *Pair programming* [69] concepts can be applied to this task. Another option is to distribute test cases among multiple people as in [62]. LEARN2FIX generates test cases by *mutational fuzzing*. As mutation operators are randomly applied in fuzzing, these test cases can be less human-readable in some scenarios. Hence, we will explore techniques to improve the human readability of the generated test cases. The works of Afshan et al. [59], McMinn et al. [60] and Bozkurt et al. [70] will be helpful for this task.

An automatic test oracle produced by LEARN2FIX expresses the condition under which a semantic bug is exposed. Hence, the automatic test oracle can serve as a specification of the bug. A bug specification describes the behaviour of a bug, and the failure condition is an essential part of it. Such a specification would be useful for developers. However, LEARN2FIX requires some improvements for producing bug specifications, as it does not always produce 100% accurate test oracles under the limited labelling queries. In future work, we will discover methods to convert LEARN2FIX for producing bug specifications.

In *constrained-based program repair* techniques, exploring the repair constraint of the given program is an important task [1][2]. The accuracy of the repair constraint determines the quality of the patch. An automatic oracle produced by LEARN2FIX consists of constraints that that explain the failure condition of a semantic bug. We will explore how to incorporate these constraints into a repair constraint used in constraint-based program repair. This will help to generate more accurate repair constraints.

10 Conclusion

We introduced LEARN2FIX, a human-in-the-loop approach, to repair programs with semantic bugs. Given a single failing input of the bug, it learns a high-quality automatic test oracle for the bug. In oracle learning, LEARN2FIX maximises the human labelling of the failing tests. In the experiments with different classifier representations, we identified that LEARN2FIX works better with *interpolating binary classifiers* than *approximating binary classifiers*. Also, the automatic oracles represented as *decision trees* are the most accurate. With both *GenProg* and *Angelix*, the auto-generated test suites in oracle learning produce better repairs compared the manual test suites of the benchmark. All these findings indicate that LEARN2FIX addresses some important problems in test oracle automation and automated program repair.

References

[1] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2017.

[2] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[3] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, "Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system," *Empirical Software Engineering*, vol. 24, pp. 33–67, 2019.

[4] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 831–841.

[5] 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*, 2018, pp. 789–799.

[6] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, Sep. 2018.

[7] B. Settles, "Active learning literature survey," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 2009.

[8] R. Longadge and S. Dongre, "Class imbalance problem in data mining review," *arXiv preprint arXiv:1305.1707*, 2013.

[9] A. Holub, P. Perona, and M. C. Burl, "Entropy-based active learning for object recognition," *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops*, pp. 1–8, 2008.

[10] M. Böhme, C. Geethal, and V.-T. Pham, "Human-in-the-loop automatic program repair," in *Proceedings of the 2020 IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST 2020, 2020, pp. 1–12.

[11] S. Kolb, S. Teso, A. Passerini, and L. De Raedt, "Learning smt (lra) constraints using smt solvers." in *IJCAI*, 2018, pp. 2333–2340.

[12] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[13] A. J. Wyner, M. Olson, J. Bleich, and D. Mease, "Explaining the success of adaboost and random forests as interpolating classifiers," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 1558–1590, 2017.

[14] V. Kecman, "Support vector machines–an introduction," in *Support vector machines: theory and applications*. Springer, 2005, pp. 1–47.

[15] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 691–701.

[16] R. Williams. (2002) Triangle classification problem. [Online]. Available: https://russcon.org/triangle_classification.html

[17] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 772–781.

[18] 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*, 2015, pp. 532–543.

[19] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[20] M. Zalewski, "American fuzzy lop," 2014.

[21] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, vol. 75, pp. 118–137, 2018.

[22] R. Braga, P. S. Neto, R. Rabêlo, J. Santiago, and M. Souza, "A machine learning approach to generate test oracles," in *Proceedings of the XXXII Brazilian Symposium on Software Engineering*. ACM, 2018, pp. 142–151.

[23] L. C. Briand, "Novel applications of machine learning in software testing," in *2008 The Eighth International Conference on Quality Software*. IEEE, 2008, pp. 3–10.

[24] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of satisfiability*. IOS Press, 2021, pp. 1267–1329.

[25] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[26] H. Jin, Y. Wang, N.-W. Chen, Z.-J. Gou, and S. Wang, "Artificial neural network for automatic test oracles generation," in *2008 International Conference on Computer Science and Software Engineering*, vol. 2. IEEE, 2008, pp. 727–730.

[27] I. H. Sarker, "Machine learning: Algorithms, real-world applications and research directions," *SN Computer Science*, vol. 2, no. 3, pp. 1–21, 2021.

[28] X. Liu, Y. Liu, Z. Li, and R. Zhao, "Fault classification oriented spectrum based fault localization," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2017, pp. 256–261.

[29] J. R. Koza *et al.*, "Evolution of subsumption using genetic programming," in *Proceedings of the first European conference on artificial life*. MIT Press Cambridge, MA, USA, 1992, pp. 110–119.

[30] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 180–182.

[31] 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," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[32] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "A feasibility study of using automated program repair for introductory programming assignments," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 740–751.

[33] M. Motwani, "High-quality automated program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2021, pp. 309–314.

[34] X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On reliability of patch correctness assessment," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 524–535.

[35] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "Code coverage," in *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023, retrieved 2023-01-07 13:54:15+01:00. [Online]. Available: https://www.fuzzingbook.org/html/Coverage.html

[36] M. Motwani, M. Soto, Y. Brun, R. Just, and C. Le Goues, "Quality of automated program repair on real-world defects," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 637–661, 2020.

[37] Y. Brun, E. Barr, M. Xio, C. Le Gouses, and P. Devanbu, "Evolution vs. intelligent design in program patching," UC Davis: College of Engineering, Tech. Rep., 2013. [Online]. Available: https://escholarship.org/2134uc/item/3z8926ks

[38] D. Dittrich, M. Bailey, and S. Dietrich, "Building an active computer security ethics community," *IEEE Security & Privacy*, vol. 9, no. 4, pp. 32–40, 2010.

[39] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[40] M. Pezze and C. Zhang, "Automated test oracles: A survey," in *Advances in computers*. Elsevier, 2014, vol. 95, pp. 1–48.

[41] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.

[42] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE transactions on Software Engineering*, vol. 21, no. 1, pp. 19–31, 1995.

[43] M. Vanmali, M. Last, and A. Kandel, "Using a neural network in the software testing process," *International Journal of Intelligent Systems*, vol. 17, no. 1, pp. 45–62, 2002.

[44] S. R. Shahamiri, W. M. N. W. Kadir, S. Ibrahim, and S. Z. M. Hashim, "An automated framework for software test oracle," *Information and Software Technology*, vol. 53, no. 7, pp. 774–788, 2011.

[45] S. R. Shahamiri, W. M. Wan-Kadir, S. Ibrahim, and S. Z. Hashim, "Artificial neural networks as multi-networks automated test oracle," *Automated Software Engg.*, vol. 19, no. 3, p. 303–334, Sep. 2012. [Online]. Available: https://doi.org/10.1007/s10515-011-0094-z

[46] A. Singh, N. Thakur, and A. Sharma, "A review of supervised machine learning algorithms," in *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*. Ieee, 2016, pp. 1310–1315.

[47] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[48] A. J. Joshi, F. Porikli, and N. Papanikolopoulos, "Multi-class active learning for image classification," in *2009 ieee conference on computer vision and pattern recognition*. IEEE, 2009, pp. 2372–2379.

[49] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[50] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980–1997, 2021.

[51] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[52] J. Röβler, G. Fraser, A. Zeller, and A. Orso, "Isolating failure causes through test case generation," in *Proceedings of the 2012 international symposium on software testing and analysis*, 2012, pp. 309–319.

[53] X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 8–18.

[54] W. Jin and A. Orso, "F3: Fault localization for field failures," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 213–223.

[55] V.-T. Pham, W. B. Ng, K. Rubinov, and A. Roychoudhury, "Hercules: Reproducing crashes in real-world application binaries," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 891–901.

[56] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test

data generation with an application to the oracle cost problem," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2010, pp. 182–191.

[57] J. Ferrer, F. Chicano, and E. Alba, "Evolutionary algorithms for the multi-objective test data generation problem," *Software: Practice and Experience*, vol. 42, no. 11, pp. 1331–1362, 2012.

[58] R. Taylor, M. Hall, K. Bogdanov, and J. Derrick, "Using behaviour inference to optimise regression test sets," in *IFIP International Conference on Testing Software and Systems*. Springer, 2012, pp. 184–199.

[59] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 352–361.

[60] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proceedings of the First International Workshop on Software Test Output Validation*. ACM, 2010, pp. 1–4.

[61] M. Staats, G. Gay, and M. P. Heimdahl, "Automated oracle creation support, or: How i learned to stop worrying about fault propagation and love mutation testing," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 870–880.

[62] F. Pastore, L. Mariani, and G. Fraser, "Crowdoracles: Can the crowd solve the oracle problem?" in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 342–351.

[63] C. Geethal, V.-T. Pham, A. Aleti, and M. Böhme, "Human-in-the-loop oracle learning for semantic bugs in string processing programs," in *Symposium on Software Testing and Analysis (ISSTA'22)*, 2022.

[64] J. F. Bowring, J. M. Rehg, and M. J. Harrold, "Active learning for automatic classification of software behavior," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 195–205, 2004.

[65] 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, pp. 1936–1964, 2017.

[66] J. Yi, S. H. Tan, S. Mechtaev, M. Böhme, and A. Roychoudhury, "A correlation study between automated program repair and test-suite metrics," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 24–24.

[67] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 471–482.

[68] S. R. Eddy, "What is bayesian statistics?" *Nature biotechnology*, vol. 22, no. 9, pp. 1177–1178, 2004.

[69] A. Begel and N. Nagappan, "Pair programming: what's in it for me?" in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, 2008, pp. 120–128.

[70] M. Bozkurt and M. Harman, "Automatically generating realistic test input from web services," in *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*. IEEE, 2011, pp. 13–24.

Charaka Geethal is a lecturer at the Department of Computer Science, Faculty of Science of the University of Ruhuna in Sri Lanka. His research interests include test oracle automation, automated program repair, automated software testing, machine learning, natural language processing and text mining. He received his Ph.D. degree in computer science in 2023 from Monash University, Australia.

Marcel Böhme is a faculty member at the Max Planck Institute for Security and Privacy (MPI-SP) in Germany where he leads the Software Security research group. He develops advanced techniques, widely used in practice, for the automatic discovery of security flaws in large software systems and works on the formal and probabilistic foundations to study the guarantees of existing approaches. He is an Associate Editor for the ACM TOSEM the flagship journal in software engineering and an Area Chair for ICSE'24, the flagship conference in software engineering. He has served on the program committees and organizational committees of all premier international conferences in software engineering. Marcel received his PhD from the National University of Singapore.

Van-Thuan Pham is a Lecturer in Cyber Security at the University of Melbourne in Australia. He received his Ph.D. degree in Computer Science from the National University of Singapore in July 2017. He has been working on scalable and high-performance fuzz testing to improve the reliability & security of software systems. His research, in collaboration with companies and government agencies, has led to many papers published at premier journals and conferences (e.g., TSE, ICSE, CCS), one U.S. patent, and one Australian provisional patent. He has developed several open-source automated security testing tools (e.g., AFLGo, AFLSmart, AFLNet, AFLTeam) that are responsible for 100+ (critical) vulnerabilities discovered in large real-world software systems. His research has been featured on media channels like Theregister.co.uk and Securityweek.com.