Empirical Software Engineering (2024) 29:_###################_
https://doi.org/10.1007/s10664-024-10503-2

Test-based patch clustering for automatically-generated patches assessment

Matias Martinez[1] · **Maria Kechagia**[2] · **Anjana Perera**[3] · **Justyna Petke**[2] · **Federica Sarro**[2] · **Aldeida Aleti**[3]

Accepted: 17 May 2024 / Published online: 24 July 2024

Abstract

Previous studies have shown that Automated Program Repair (APR) techniques suffer from the overfitting problem. Overfitting happens when a patch is run and the test suite does not reveal any error, but the patch actually does not fix the underlying bug or it introduces a new defect that is not covered by the test suite. Therefore, the patches generated by APR tools need to be validated by human programmers, which can be very costly, and prevents APR tool adoption in practice. Our work aims to minimize the number of plausible patches that programmers have to review, thereby reducing the time required to find a correct patch. We introduce a novel light-weight test-based patch clustering approach called xTestCluster, which clusters patches based on their dynamic behavior. xTestCluster is applied after the patch generation phase in order to analyze the generated patches from one or more repair tools and to provide more information about those patches for facilitating patch assessment. The novelty of xTestCluster lies in using information from execution of newly generated test cases to cluster patches generated by multiple APR approaches. A cluster is formed of patches that fail on the same generated test cases. The output from xTestCluster gives developers *a)* a way of reducing the number of patches to analyze, as they can focus on analyzing a sample of patches from each cluster, *b)* additional information (new test cases and their results) attached to each patch. After analyzing 902 plausible patches from 21 Java APR tools, our results show that xTestCluster is able to reduce the number of patches to review and analyze with a median of 50%. xTestCluster can save a significant amount of time for developers that have to review the multitude of patches generated by APR tools, and provides them with new test cases that expose the differences in behavior between generated patches. Moreover, xTestCluster can complement other patch assessment techniques that help detect patch misclassifications.

Keywords Automated program repair · Patch assessment · Patch overfitting · Test case generation · Java

Communicated by: Jinqiu Yang

Extended author information available on the last page of the article

1 Introduction

Automated program repair (APR) techniques generate patches for fixing software bugs automatically (Gazzola et al. 2017; Monperrus 2018). The aim of APR is to significantly reduce the manual effort required by developers to fix software bugs. However, it has been shown that APR techniques tend to produce more incorrect patches than correct ones (Le et al. 2018; Long and Rinard 2016; Martinez et al. 2017). This issue is also known as the *overfitting* (or test-suite-overfitting) problem. Overfitting happens when a patch generated automatically passes all the existing test cases yet it fails in presence of other inputs which are not captured by the given test suite (Smith et al. 2015). This happens because the test cases, which are used as program specification to check whether the generated patches fix the bug may be insufficient to fully specify the correct behavior of a program. As a result, a generated patch may pass all the existing tests (i.e., the patch can be a *plausible* fix), but still be incorrect (Qi et al. 2015).

Due to the overfitting problem, developers have to *manually assess* the generated patches before integrating them to the code base. Manual patch assessment is a very time-consuming and labor-intensive task (Ye et al. 2021), especially when multiple plausible patches are generated for a given bug (Le et al. 2018; Martinez and Monperrus 2018). To alleviate this problem, different techniques for *automated patch assessment* have been proposed. Filtering of overfitted patches can happen during patch generation, as part of the repair process (e.g. Long and Rinard (2016)), or as part of the post-processing of the generated patches (e.g. Ye et al. (2021); Xiong et al. (2018)). Typically, such techniques focus on the prioritization of patches. The patches ranked at the top are deemed to be the most likely to be correct. Existing approaches often rank similar patches at the top (Le et al. 2018) and as a result waste developers' time if the top-ranked patches are overfitted. Furthermore, such approaches might require an oracle (Xin and Reiss 2017), or (often expensive) program instrumentation (Xiong et al. 2018), or a more sophisticated machine learning process (Ye et al. 2021).

To alleviate these issues, we present a light-weight patch post-processing technique, named xTestCluster, that aims to reduce the number of generated patches that a developer has to assess. Our technique clusters plausible repair patches exhibiting the same behavior (according to a given set of test suites), and provides the developer with fewer patches, each representative of a given cluster, thus ensuring that those patches exhibit different behavior. Our technique can be used not only when a single tool generates multiple plausible patches for a given bug, but also when different available APR tools are running (potentially in parallel) in order to increase the chance of finding a correct patch. In this way, developers will only need to examine one patch, representative of a given cluster, rather than all, possibly hundreds, of patches produced by APR tools.

Our approach presents two main novelties: First, it leverages the diversity of the behavior of the generated patches (and this diversity is not exposed by the developer-written test cases used to synthesize patches). In particular, our clustering approach xTestCluster exploits automatically generated test cases that enforce diverse behavior in addition to the existing test suite, as opposed to previous work (including (Mechtaev et al. 2018) for patch generation and Cashin et al. (2019) for patch assessment) that use solely the existing test suite written by developers. Second, our approach has the main advantage that it does not involve code instrumentation (aside from patch application) nor an oracle (e.g., Xin and Reiss (2017)) or pre-existing dataset to learn fix patterns (e.g., Ye et al. (2021); Lin et al. (2022)). Moreover, xTestCluster is complementary to previous work on patch overfitting assessment, as it can apply different prioritization strategies to each cluster.

xTestCluster works as follows: First, xTestCluster receives as input a set of plausible patches generated by a number of selected apr tools and it generates new test cases for the patched files (i.e., buggy programs to which plausible patches have been applied) using automated test-case generation tools. The goal of this step is to generate new inputs and assertions that expose the behavior (correct and/or incorrect) of each generated patch. Second, xTestCluster executes the generated test cases on each patched file to detect any behavioral differences among the generated patches (we call this step *cross test-case execution*). Third, xTestCluster receives the results from the execution of each test case on a patched version of a given buggy program, and uses the names of the failing test cases to cluster patches together. In other words, patches from the same cluster exhibit the same output, according to the generated test cases: they fail on all the same generated tests. Patches that pass all the test cases (no failing tests) are clustered together.

We evaluate our approach on 902 patches (248 correct and 654 overfitted) for bugs from v.1.5.0 of the Defects4J data set (Just et al. 2014), generated by 21 different apr tools, and collected and labeled by Wang et al. (2020). After removing duplicates, we used two automated test-case generation tools, EvoSuite (Fraser and Arcuri 2011) and Randoop (Pacheco and Ernst 2007), to generate test cases for our patch set. Finally, we cluster patches based on test case results. To our knowledge, xTestCluster is the first approach to analyze together patches from multiple program repair approaches generated to fix a particular bug. This is important because it shows that xTestCluster can be used in the wild, independently of the adopted Java repair tools.

Our results show that xTestCluster is able to create at least two clusters for almost half of the bugs that have two or more different patches. By having patches clustered, xTestCluster is able to reduce a median of 50% of the number of patches to review and analyze. This reduction could help code reviewers (developers using automated repair tools or researchers evaluating patches) to reduce the time of patch evaluation. Additionally, xTestCluster can also provide code reviewers with the inputs (from the generated test cases) that trigger different program behaviors for different patches generated for one bug. This additional information may help them decide which patch to select and merge into their codebase.

We also analyze the assessment done by two state-of-the-art patch assessment approaches, ODS (Ye et al. 2021) and Cache (Lin et al. 2022) on the patches clustered by xTestCluster. The results show that xTestCluster can be used complementarily to those approaches and can help to detect false positives and false negatives. Finally, we study whether metrics related to the quality of the newly-generated tests in order are related to the efficiency of xTestCluster. We found that the number of test cases generated, their lengths (in terms of lines of code), and the coverage affect the ability to cluster correct patches together.

Overall, the paper provides the following contributions:

- A novel test-based patch clustering approach called xTestCluster. It is complementary to existing patch overfitting assessment approaches. xTestCluster can be applied to patches generated by multiple APR tools.
- An implementation of xTestCluster for analyzing Java patches. It uses two popular automated test-case generation frameworks, EvoSuite (Fraser and Arcuri 2011) and Randoop (Pacheco and Ernst 2007). The code of xTestCluster is publicly available (xtestcluster appendix repository 2021).
- An evaluation of xTestCluster using patches from 21 apr tools, and 920 plausible patches.

All our data is available in our appendix (xtestcluster appendix repository 2021).

2 Our approach

Our proposed approach, xTestCluster, for test-based patch clustering is shown in Fig. 1.

Algorithm 1 xTestCluster.

```
1: Input: $B$ a buggy program, $Ps$ plausible patches of bug $B$, $TGs$ test-case generators.
2: $TCG \leftarrow testGeneration(B, Ps, TGs)$ (Alg. 2)
3: $ResPatches_{exec} \leftarrow testExecution(B, Ps, TCG)$ (Alg. 3)
4: $clusters, exec\_info \leftarrow clustering(Ps, ResPatches_{exec})$ (Alg. 4)
5: return $clusters, exec\_info$
```

xTestCluster receives as input a buggy program and a set of plausible patches that could repair the bug. The patches could have been automatically generated by one or multiple repair approaches. Additionally, xTestCluster receives a set of test-case generation tools. Given these inputs, xTestCluster carries out three steps (lines 2–5 of Algorithm 1): *1)* test-case generation, *2)* test-case execution, and *3)* clustering.

Test-case generation xTestCluster receives as input the plausible patches generated by a set of APR tools and generates new test cases for the patched files (i.e., buggy programs to which plausible patches were applied to). We use automated test case generation tools for this purpose.

Test-case execution xTestCluster executes the generated test cases on each patched version of the program: We call this approach *cross test-case execution*, because the test cases generated for a given patched program version are executed on another patched version

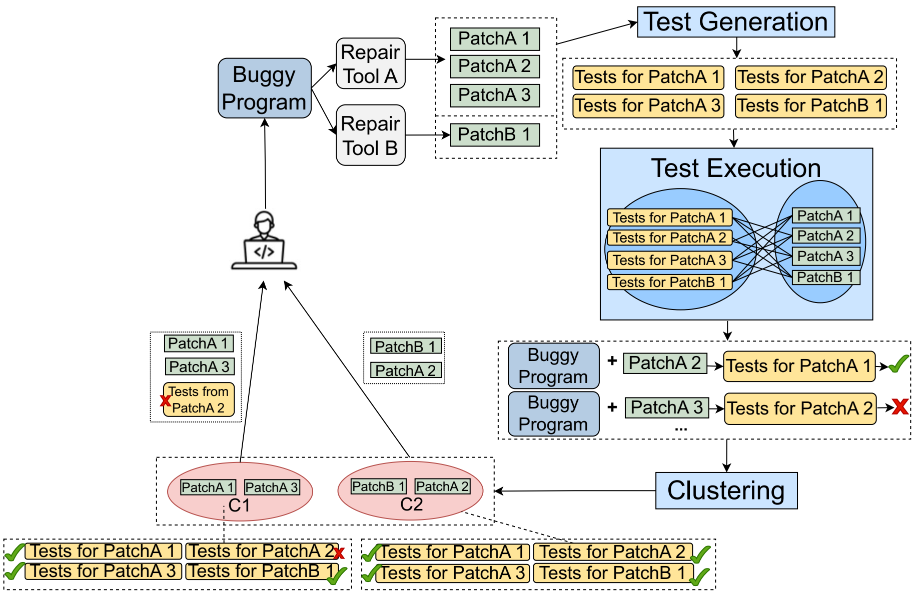


Fig. 1 The steps executed by xTestCluster

Algorithm 2 Test-case Generation.

```
1: Input: $B$ a buggy program, $Ps$ plausible patches of bug $B$, $TGs$ test-case generators.
2: $TCG \leftarrow \emptyset$
3: for $patch \in Ps$ do
4:     $B' \leftarrow$ apply $patch$ to $B$
5:     $pfiles \leftarrow getFiles(patch)$
6:     for $tg \in TGs$ do
7:         for $pfile \in pfiles$ do
8:             $TC_{new} \leftarrow generateTests(tg, B', pfile)$
9:             $TCG \leftarrow TCG \cup TC_{new}$
10:        end for
11:    end for
12: end for
13: return $TCG$
```

of a given buggy program. This cross execution aims to detect the behavioral differences among the generated patches that we use in clustering afterward.

Clustering xTestCluster receives the results from the execution of each test case on a patched version of a given buggy program and uses this information to cluster patches together: if two patches have the same output for all the test cases generated, then they belong to the same cluster. Each cluster of patches will also be furnished with test cases for which the patches fail, and the corresponding failures observed.

These steps allow xTestCluster to reduce the number of patches that are presented to the *code reviewer*. A code reviewer could be, for example, a software developer that has developed and pushed a buggy version, which is exposed, for example, through failed test cases executed by a continuous integration platform (CI). Without using xTestCluster, the code reviewer needs to assess patches produced by repair approaches that they have integrated, for example, in their CI. Using xTestCluster, the code reviewer can, now, review a *subset* from all the generated patches, reducing the review effort. Moreover, for each presented patch, they also have alternative patches (those not selected from the same cluster but with the same behavior as the selected patch) and information about test-case executions. All this information could help code reviewer decide which patch to integrate into the codebase to fix the given bug.

In the following subsections, we detail each step of xTestCluster.

2.1 Test-case Generation

Algorithm 2 details this step. For each patch *patch* from those plausible patches received as input (line 3), xTestCluster first applies this patch to the buggy program B, giving the patched program B' as a result (line 4). We recall that the patched program must pass all test cases provided by the developer. If the patch does not pass any of those test cases, it is not plausible, and xTestCluster discards it. Then, xTestCluster retrieves the files affected by the patch (line 5). Using each of the test-case generation tools (line 6) we have selected, xTestCluster generates test cases for each of those files that have plausible fixes (line 8). All the generated test cases are stored in a set called TCG (line 9). xTestCluster is agnostic to the test case generation tools employed in practice. This means that, in theory, any test case generation tool could be used. For this reason, in this section we do not detail the implementation of the $generateTest$ invocation at line 8, i.e., how tests are generated.

Algorithm 3 Test-case Execution.

```
1: Input: $B$ a buggy program , $Ps$ plausible patches of bug $B$, $TCG$ test cases generated.
2: $ResPatches_{exec} \leftarrow \emptyset$
3: for $patch \in Ps$ do
4:     $R_{exec} \leftarrow \emptyset$
5:     $B' \leftarrow$ apply $patch$ to $B$
6:     for $t \in TCG$ do
7:         $res_t \leftarrow execute(t, B', patch)$
8:         $R_{exec} \leftarrow R_{exec} \cup (res_t, patch)$
9:     end for
10:    $ResPatches_{exec} \leftarrow ResPatches_{exec} \cup \langle patch, R_{exec} \rangle$
11: end for
12: return $ResPatches_{exec}$
```

Nevertheless, in the Methodology Section 4.2 we detail implementation of our approach used in the evaluation (which is based on two state-of-the-art test generation tools: Evosuite (Fraser and Arcuri 2011) and Randoop (Pacheco and Ernst 2007)).

xTestCluster also carries out a *sanity check* on the generated test cases. In particular, it verifies that they are not flaky by executing them n times, and assuring that the results are the same for each execution. Test cases that do not pass this check are discarded.[1]

2.2 Test-case execution

Next, we conduct *cross test-case execution*. Algorithm 3 details this step. xTestCluster executes, on a version of the program patched with the patch $patch$, the test cases generated by considering other plausible patches for bug B. In other words, the step applies the Cartesian product between patches and test cases produced for the patches. To achieve this, xTestCluster iterates over the patches (line 3). For each patch, xTestCluster applies it to the buggy program, producing the patched program B' as a result (line 5). xTestCluster executes each new test case t generated in the previous step (set from TCG) over the patched program B' (lines 6 and 7). All results from the test-case execution for $patch$ are stored in the map $ResPatches_{exec}$ (line 10), which is then returned (line 12).

2.3 Clustering

Algorithm 4 details this step. xTestCluster iterates over the patches (line 4) in order to assign each patch $patch_i$ to a cluster. If no cluster has been previously created (line 5), xTestCluster creates a new cluster that includes $patch_i$ (line 6), and stores the results of the test cases (i.e., the failing test cases if any) in the list $FailsCs$ (line 7). The i-th element of that list contains the execution results of the i-th cluster. Otherwise, xTestCluster first retrieves the results from the *test-case execution* step (Algorithm 3) for that patch (line 9). Then, xTestCluster iterates over the created clusters (line 11). For each $cluster$, xTestCluster chooses a patch $patch_o$ from it (line 12) and retrieves the corresponding results from the test case execution step (line 13). xTestCluster compares the two execution results (line 14): If both patches produce the same failures in our test set after being applied to the buggy program, the patch $patch_i$ is included in $cluster$ (line 15). Note that the result of a test case can be *passing* or *failing*. When the result is failing, we also consider the message associated

[1] For simplicity we do not show this check in Algorithm 2.

Algorithm 4 Clustering.

```
1: Input: $Ps$ plausible patches of bug $B$, $ResPatches_{exec}$ results of test cases.
2: $Cs \leftarrow \emptyset$
3: $FailsCs \leftarrow \emptyset$
4: for $patch_i \in Ps$ do
5:     if $Cs$ is $\emptyset$ then
6:         $Cs \leftarrow set(patch_i)$
7:         $FailsCs \leftarrow getTestExecution(ResPatches_{exec}, patch_i)$
8:     else
9:         $R_{exec_i} \leftarrow getTestExecution(ResPatches_{exec}, patch_i)$
10:        $foundCluster \leftarrow false$
11:        for $cluster \in C_s$ do
12:            $patch_o \leftarrow getOne(cluster)$
13:            $R_{exec_o} \leftarrow getTestExecution(ResPatches_{exec}, patch_o)$
14:            if $R_{exec_i} = R_{exec_o}$ then
15:                $cluster \leftarrow cluster \cup patch_i$
16:                $foundCluster \leftarrow true$
17:                break
18:            end if
19:        end for
20:        if foundCluster = false then
21:            $Cs \leftarrow Cs \cup set(patch_i)$
22:            $FailsCs \leftarrow FailsCs \cup R_{exec_i}$
23:        end if
24:    end if
25: end for
26: return $Cs$, $FailsCs$
```

with the failing assertion or the message associated with an error. Consequently, patches that do not pass a test case due to different reasons (e.g., some fail an assertion and others produce a `null` pointer exception) are allocated into different clusters. Patches that pass all test cases (this means that R_{exec_o} at line 13 is empty) are clustered together. If xTestCluster cannot allocate $patch_i$ to any cluster (line 20), it creates a new cluster which includes $patch_i$ (line 21) and stores the test execution result in $FailsCs$ (line 22).

Finally, xTestCluster returns all the created clusters Cs and a list with the test case execution (i.e., failing cases) for each of those clusters (line 26).

3 Research questions

Our proposal, xTestCluster, aims to aid code reviewers in reducing the effort required for manual assessment of patches automatically generated by APR tooling. In order to assess how well xTestCluster can achieve this task we pose the following RQs:

RQ1: Hypothesis Validation *To what extent are generated test cases able to capture behavioral differences between patches generated by* APR *tools?*

This RQ aims to show the ability of xTestCluster to detect behavioral differences among the generated patches based on the execution of generated test cases, and, from those differences, to create clusters of patches. If successful, semantically equivalent patches will be clustered together and only one, thus, needs to be presented to code reviewer from such a cluster.

RQ2: Patch Reduction Effectiveness *How effective is* xTestCluster *at reducing the number of patches that need to be manually inspected?*

This RQ aims to show the applicability of xTestCluster in order to help developers reduce the effort of manually reviewing and assessing patches. In particular, we compare the number of patches produced by all selected APR tools vs. the number of patches presented to code reviewer if our approach is used.

RQ3: Clustering Effectiveness *How effective is* xTestCluster *at clustering correct patches together?*

This RQ aims to measure the ability of xTestCluster to cluster correct patches together. In case each cluster contains *only* either correct or incorrect patches, we can simply pick *any* patch from a given cluster for validation. In other words, picking *any* patch from a cluster would be sufficient to ensure existence (or non-existence) of a correct patch among *all* plausible patches in a cluster during patch assessment.

RQ4: Complementing State-of-the art patch assessment techniques *To what extent is* xTestCluster *able to complement existing overfitting detection techniques in order to help them to reduce the rate of incorrect assessments?*

This RQ aims to study the assessment done by state-of-the-art patch assessment approaches on the clusters found by xTestCluster. We will specially focus on mixed clusters, i.e. those that contain both correct and incorrect patches, and we study the false positives and false negatives of other patch assessment approaches on patches assigned by xTestCluster to those clusters.

RQ5: quality of generated test cases and effectiveness of xTestCluster *To what extent does quality of the newly-generated test cases affect* xTestCluster *'s effectiveness?*

This RQ aims to study the relationship between the quality of test cases (represented using metrics extracted from the newly generated test cases) and the effectiveness of xTestCluster to differentiate correct patches from incorrect. In particular, we want to know if the limitations of xTestCluster are due to the quality of the generated tests it uses for comparing the behaviours.

4 Methodology

In this section, we present the methodology followed to answer our research questions.

4.1 Dataset

In order to evaluate xTestCluster we need a set of plausible patches, i.e., proposed fixes for a given bug. There are two constraints the dataset needs to meet. Firstly, as xTestCluster focuses on the reduction of the amount of patches to be presented to the developers for review, xTestCluster makes sense only if there are at least two different plausible patches for a given bug. Secondly, we need to know whether each patch in the dataset is correct or not. In previous work (e.g., Ye et al. (2021)) patches have been labelled (usually via manual analysis) as either *correct*, *incorrect* (or *overfitting*), or marked as *unknown* (or *unassessed*).

We will use the *correct* and *incorrect* label terminology, and only consider patches for which correctness has been established.

We thus consider patches generated by existing repairs tools. In this experiment, we focus on tools that repair bugs in Java source code because: 1. Java is a popular programming language, which we aim to study, and 2. the most recent repair tools have been evaluated on bugs from Java software projects.

Since the execution of APR tools to generate patches is very time-consuming (especially if we consider several repair tools (Durieux et al. 2019)), we use publicly available patches that were generated in previous APR work. We also decide to rely on external patch evaluations done by other researchers and published as artifacts to peer-reviewed publications. This avoids possible researcher bias. Furthermore, it allows us to gather a large dataset of patches, from multiple sources, and avoid the costly manual effort of manual patch evaluation of thousands of patches.

Taking all constraints into account, we decided to study patches for bugs from the DEFECTS4J (Just et al. 2014) dataset. To the best of our knowledge, DEFECTS4J is the most widely used dataset for the evaluation of repair approaches (Martinez et al. 2017; Durieux et al. 2019; Liu et al. 2020). Consequently, hundreds of patches for fixing bugs in DEFECTS4J are publicly available. We leverage data from previous work that has collected and aggregated patches generated by different Java repair tools, all evaluated on DEFECTS4J. We focus on those that, in addition, provide a *correctness label*.

In this paper, we use the dataset provided by Wang et al. (2020) which contains 902 patches from 21 repair systems. We choose this dataset for the following main reasons.

- Firstly, it met all our criteria mentioned above (i.e., Java patches, Defects4J patches, correctness labels).
- Secondly, it is a combination of other two datasets of labeled patches, one from Liu et al. (2020) and the second from DDR by Ye et al. (2021). Liu et al. (2020) included patches from 16 repair systems, and manually evaluated the correctness using guidelines presented by Liu et al. (2020). Ye et al. (2021) classified patches using a technique called RGT, which generates new test cases using ground-truth, human-written oracle patches. The patches from these datasets were revised by Wang and colleagues in order to correct existing correctness label misclassifications.
- Lastly, the dataset from Wang et al. has been widely used in the evaluation of overfitting approaches, including (Ye et al. 2021; Lin et al. 2022; Wang et al. 2020). Using this dataset allows us to compare the performance of xTESTCLUSTER with previous work.

Table 1 presents the number of patches from the Wang et al. dataset per repair tool. The dataset contains 902 bugs, 248 were labeled as correct, and the other 654 as overfitting (incorrect).

We now explain how we processed the Wang et al. datasets in order to select the bugs and patches that we are interested in analyzing. Table 2 presents the number of bugs for which patches were generated. In total, 202 bugs from the version 1.5.0 of DEFECTS4J contain at least one patch on Wang et al.'s dataset.

For each bug, we carry out a syntactic analysis of patches (using the *diff* command) in order to detect duplicate patches produced by two or more repair tools. This is necessary, as multiple tools can create exactly the same patch. In total, we consider 777 distinct patches.

From the patches gathered for 202 bugs in our dataset, we find that for 70 bugs there is only one single patch. We discard those bugs and their patches because xTESTCLUSTER needs at least two patches per bug. We also discard patches for three further bugs for the following reasons: First, bugs Closure-63 and Closure-93 were deprecated in a recent version

Table 1 Number of Correct and Overfitted patches for bugs from DEFECTS4J (Just et al. 2014) per repair tool, contained in the dataset from Wang et al. (2020) composed of 902 patches

Tools	Dataset of patches	
	Wang et al. (2020)	
	Correct	Overfitted
ACS (Xiong et al. 2017)	15	7
AVATAR (Liu et al. 2018)	19	38
Arja (Yuan and Banzhaf 2018)	5	52
CapGen (Wen et al. 2018)	25	41
Cardumen (Martinez and Monperrus 2016)	2	10
DynaMoth (Durieux and Monperrus 2016)	1	21
FixMiner (Koyuncu et al. 2020)	12	20
GenProg-A (Yuan and Banzhaf 2018)	2	28
Jaid (Chen et al. 2017)	40	41
jGenProg (Martinez and Monperrus 2016)	3	17
jKali (Martinez and Monperrus 2016)	3	22
jMutRepair (Martinez and Monperrus 2016)	5	17
Kali-A (Yuan and Banzhaf 2018)	3	60
kPAR (Liu et al. 2019)	10	52
Nopol (Xuan et al. 2017)	1	30
RSRepair-A (Yuan and Banzhaf 2018)	2	39
SOFix (Liu and Zhong 2018)	21	2
SequenceR (Chen et al. 2019)	17	56
SimFix (Jiang et al. 2018)	22	46
SketchFix (Hua et al. 2018)	16	7
TBar (Liu et al. 2019)	24	48
Total	248	654

of Defects4J.[2] Consequently, the Defects4J framework does not allow us to checkout and test those bugs. Second, for Math-35 bug, we could generate tests for only one patch, so we discarded it.

In total, we evaluate XTESTCLUSTER on 129 bugs, each having at least two patches which we can successfully generate test cases for.

4.2 RQ1: Hypothesis validation

In order to answer RQ1, we group patches by bug repaired by the tools. Then, we apply the algorithm described in Section 2.1. In this experiment, we report the results obtained by using Evosuite (Fraser and Arcuri 2011) as the test-case generation tool. The results by adding test cases from Randoop (Pacheco and Ernst 2007) are discussed in Section 6. For each bug from DEFECTS4J, we generate test cases for each patched version, i.e., after applying a candidate patch to the buggy program, using both tools and a time budget of 60 seconds for the test-case generation[3].

[2] https://github.com/rjust/defects4j#the-projects

[3] In the Evosuite code, the default search budget is set to 60 seconds (https://github.com/EvoSuite/evosuite/blob/1948d763944b3c3275d9564cf375b31981aedab0/client/src/main/java/org/evosuite/Properties.java#

Table 2 Summary of bugs from DEFECTS4J and their respective patches collected from the three datasets

Summary of Bugs	#Bugs
Total bugs from DEFECTS4J (Just et al. 2014)	375
Total bugs with labelled patches	202
Bugs with one patch	70
Bugs with $>$ distinct one patch	132
Bugs considered by XTESTCLUSTER	129

As this approach aims to not depend on any oracle (inc. human-written test cases), we trigger the test generation from scratch, without providing any test cases as seed. When invoking these generation tools, XTESTCLUSTER uses the default values for each hyperparameter[4]. In particular, in Evosuite, the default objective of the search is to maximize the line coverage of the test cases under generation. Additionally, by default, EvoSuite applies minimization to generated test cases, which means that it removes all statements that are not strictly needed to satisfy the coverage goals.

Then, we execute the test cases generated on the patched versions (cross test-case execution). Finally, we cluster patches for a single bug by putting together all the patches that have the same results on the generated test cases, as explained in Section 2.3.

4.3 RQ2: Patch Reduction Effectiveness

To answer RQ2 we take as input the number of clusters generated in RQ1 and the total number of patches in our dataset per bug. We recall that these patches from a bug could come from: *1)* a single repair tool (which does not stop the search after the first test-passing patch is found and finds multiple such patches), *2)* multiple repair tools (where each could potentially stop after the first test-passing patch is found).

For each bug, we compute the reduction of patches to analyze per bug B as follows:

$$reduction(B) = \frac{(\#patches\ for\ B - \#clusters\ for\ B)}{\#patches\ for\ B} \times 100 \tag{1}$$

This gives us the % reduction of patches presented to a code reviewer. Recall that we only present one patch from each cluster to code reviewer, i.e, $\#clusters$ patches. Otherwise, code reviewer would have had to review $\#patches$ per bug.

4.4 RQ3: clustering effectiveness

To answer RQ3, we take as input: *1)* the clusters generated by XTESTCLUSTER (we recall a cluster has one or more patches), and *2)* the correctness labels for each patch in our dataset (see Section 4.1). We say that a cluster is *pure* if all its patches have the same label, i.e, all patches are correct or all patches are incorrect. Otherwise, we say the cluster is *not pure*.

L690), however we found an official documentation mentioning a different value (10 minutes, https://www.evosuite.org/documentation/commandline/). To avoid any confusion, we call Evosuite by explicitly passing the search budget of 60 seconds.

[4] Default values of Evosuite parameters https://github.com/EvoSuite/evosuite/blob/1948d763944b3c3275d9564cf375b31981aedab0/client/src/main/java/org/evosuite/Properties.java#L516

For each of the sets of patches per bug, we compute the ability of xTestCluster to generate only pure clusters. Having bugs with only pure clusters is the main goal of xTestCluster: If all patches in a cluster are correct, by picking one of them we are sure to present a correct one to the code reviewer. Similarly, if all patches from a cluster are incorrect, by picking one of them we are sure to present to the code reviewer an incorrect patch. In both cases, the reduction of patches presents no risk and patches can be picked from a cluster in any order, e.g., at random.

4.5 RQ4: Complementing State-of-the art patch assessment techniques

To answer RQ4, we study the performance of patch assessment approaches in patches clustered by xTestCluster. For a fair comparison, we consider approaches also evaluated on the same data used in this paper (Wang et al. (2020)). The recent empirical study by Lin et al. (2022) compared 15 patch assessment approaches (including PatchSim (Xiong et al. 2018), S3 (Le et al. 2017), CapGen (Wen et al. 2018), ssFix (Xin and Reiss 2017), ODS (Ye et al. 2021) and Cache (Lin et al. 2022)) on the mentioned dataset. They show that ODS and Cache are the approaches with the highest *accuracy*, *recall* and *f1* metrics. As their values of *f1* are extremely similar, we decided to include both of them in this experiment.

We first observe their results on patches assigned to pure clusters. That will help us identify the cases for which xTestCluster works as expected but any of the state-of-the-art approach fails. Then, we study their performance on patches assigned to mixed clusters. That is, on the cases for which xTestCluster fails. We study whether state-of-the-art approaches also fail in such cases. As both ODS and Cache have been evaluated on the Wang et al. dataset, composed of 902 patches, we consider the performance of these tools available in the Appendix of ODS[5] and Cache[6].

4.6 RQ5: Quality of generated test cases and effectiveness of xTestCluster

To answer RQ5, we consider four metrics associated with the newly generated test cases. Those metrics are: a) number of test cases generated per patch, b) lines of code corresponding to these test cases, c) line coverage that reach these test cases i.e., the number of executed lines per these test cases divided by the total lines of code contained in them, and d) mutation score i.e., total number of mutants killed by these test cases divided by the total number of mutants evaluated.

To know to what extent these metrics are related to the xTestCluster effectiveness we carry out the following steps. For each of the metrics, we first create two samples: The first one contains the metric values of the test cases that allow xTestCluster to create pure clusters (in other words, those are the tests generated from patches assigned to pure clusters). The second one contains the metric values of those test cases that are not capable of differentiating the behaviour between correct and incorrect patches (in other words, those are the tests generated from patches that belong to mixed clusters). Then, we compare the two samples to know whether the samples have the same distribution (Null hypothesis H_0) or different distribution (Alternative hypothesis). If the Null hypothesis holds, it means that the metric does not have an impact on xTestCluster effectiveness. To test the hypotheses, we use the Mann-Whitney U test, with a significance cutoff (α level) equal to 0.05 (5%).

[5] ODS appendix: https://github.com/SophieHYe/ODSExperiment

[6] Cache appendix: https://github.com/Ringbo/Cache

5 Results

In this section we present the results of our experiments with answers to our research questions.

5.1 RQ1: Hypothesis validation

As Table 3 shows, xTestCluster is able to generate more than one cluster for 78 bugs (60.4%) containing more than one plausible patch (129 bugs in total). This means that xTestCluster, using generated test cases, is able to differentiate between patches whose application produces different behavior.

For 51 bugs (39.6%), for which we have more than two syntactically different patches, xTestCluster groups all of them into one cluster. We conjecture that this could be caused by the following reasons: *1)* Beyond syntactical differences, the patches could be semantically equivalent; *2)* test-case generation tools are not able to find inputs that expose behavioral differences between the patches; *3)* test-case generation tools are not able to find the right assertion for an input that could expose behavioral differences.

Figure 2 shows the distribution of the number of clusters that xTestCluster is able to create per bug (in total, 129 bugs as explained in Section 4.2). We observe that the distribution is right-skewed. The most left bar corresponds to the previously mentioned 51 bugs with one cluster. Then, the number of bugs with *n* clusters decreases as *n* increases. For 76 bugs, xTestCluster generated between two and six clusters, but for 2 bugs, it generates a larger number of clusters (eight and fifteen).

Answer to RQ1. Given 129 bugs with at least two plausible and syntactically different patches, for 78 of them (59.4%), xTestCluster is able to detect patches with different behavior (based on test-case generation) and group them into different clusters.

By reviewing one patch per cluster, a code reviewer can reduce the time and effort required for reviewing, since he or she does not need to review all the generated patches for 59.4% of the bugs (78 in total). In particular, let us focus on these 78 bugs for which xTestCluster reduces the number of patches. Without xTestCluster, the mean number of patches to be reviewed by a developer is 6.89 patches. On the contrary, xTestCluster offers a developer a median of 3.26 patches, all of which behave differently according to the generated test cases. This means that xTestCluster actually helps to reduce the number of patches required to be reviewed.

Table 3 RQ1. Classification of bugs according to the number of patches and clusters generated by xTestCluster

Bugs under analysis	#Bugs
Bugs with one or more patches	202
Bugs with 1+ syntactically different patches	129
Bugs with multiple patches in one single cluster	51
Bugs with multiple patches and clusters	78

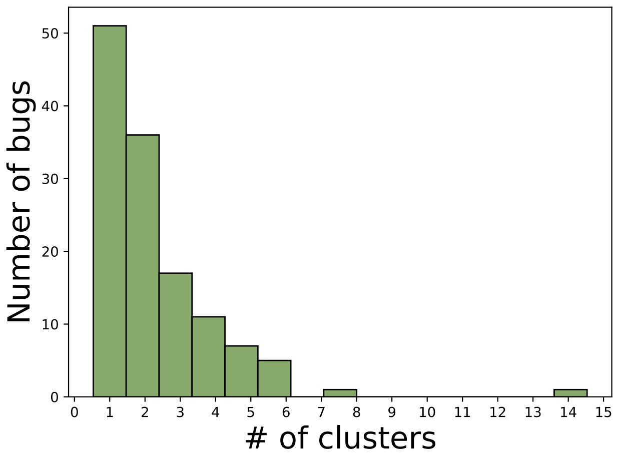


Fig. 2 RQ1. Distribution of the number of clusters per bug. Bugs with a single patch (in total 51) are discarded

5.2 RQ2: Patch Reduction Effectiveness

We compare the number of patches produced by all selected APR tools vs. the number of patches presented to code reviewer if our approach is used. Figure 3 shows the distribution of the number of patches per bug to analyze without and with our approach (red and blue bars, respectively). We observe that the distribution of patches using our tool (in blue) is distributed more to the left than the other (in red). More cases on the left mean that the number of patches to analyze is fewer.

Now, we focus on each bug: we study the percentage reduction in the patches to be analyzed when xTESTCLUSTER is used vs. reviewing all available patches for a given bug.

The median percentage reduction is 50% (46.98%), which means that for half of the bugs xTESTCLUSTER reduces the total number of patches one needs to analyze by at least 50%.

The distribution in Fig. 4 also shows that for 23 bugs we achieve no reduction. That is, for 23 bugs each cluster contains a single patch, thus all need to be analyzed. For most of

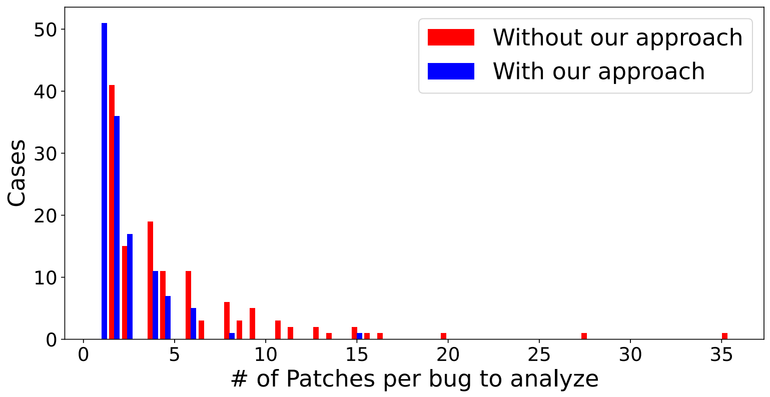


Fig. 3 RQ2. Distribution of patches to analyze per bug with and without our approach

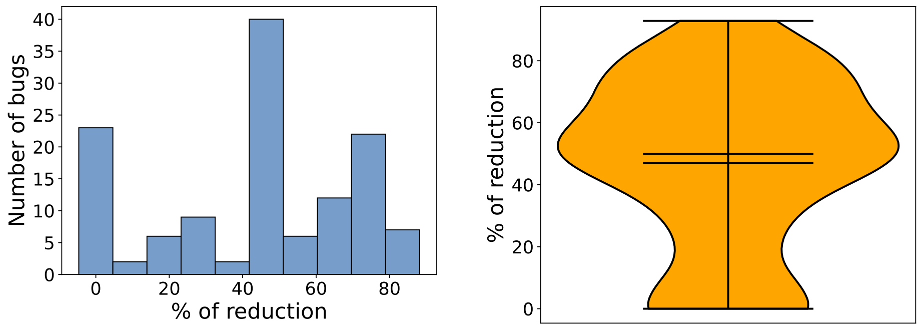


Fig. 4 RQ2. Distribution of the percentage reduction of the number of patches to review. Reduction for a bug B is computed as follows: ((#patches for B - #clusters for B) / #patches for B) × 100

those cases (16), the number of patches (and clusters) is two. In other words, a code reviewer only needs to check two patches per bug. Even in the cases there is no reduction in terms of number of patches to be analyzed, xTestCluster provides developers, for each of those patches, new inputs (included in the generated test cases) that expose the unique behavior of each of those patches. Using that information, our approach could help developers decide which patch is better.

Answer to RQ2. xTestCluster is able to reduce by a median of 50% the number of patches to analyze per bug. Thus xTestCluster could help code reviewers (developers using repair tools or researchers evaluating patches) to reduce the time required for patch assessment.

The findings discussed thus far already show that our approach could be very useful to code reviewers. Firstly, it can significantly reduce the number of patches for review, thus reducing the time and effort required for this task. Secondly, it can provide code reviewers with test inputs that help differentiate between patches, thus reducing the complexity of patch review.

Execution time of xTestCluster We calculate the execution time of xTestCluster for a given bug by summing the time required for two tasks: *1)* generating tests for each patch discovered for a given bug (limited to a one-minute time budget), and *2)* executing the newly generated tests for each discovered patch. This results in a total number of executions equivalent to the product of the number of generated tests and the number of discovered patches for a given bug.

The median execution time per bug is 4.06 minutes (avg. 5.63). We recall that the process of finding candidate patches is time-consuming (Martinez et al. 2017; Liu et al. 2020, 2021), taking more than this time to find the first plausible patches, as it requires several compilations and validation using human-written test cases. Consequently, the overhead introduced by xTestCluster does not significantly impact the time of the repair process. Nevertheless, in the Discussion (Section 6) we discuss potential solutions to further decrease the execution time of xTestCluster.

5.3 RQ3: clustering effectiveness

5.3.1 Analysis of bugs with multiple clusters

We now focus on the 78 bugs for which our approach produced multiple clusters. We analyze their purity, i.e., if the patches in a given cluster are all correct or all incorrect. This will allow us to measure the ability of test-case generation to differentiate between correct and incorrect patches. Table 4 summarizes our results. The first two rows group the bugs that have both correct and incorrect patches (17 + 21 = 38 in total). The third and fourth rows show the number of bugs with either only incorrect patches (40 in total), or only correct patches (zero in total, which means that for every correct patch there is at least one repair tool that generates an incorrect patch).

For 17 out of the 38 bugs (44%) with both correct and incorrect patches, all clusters generated by xTestCluster are pure. This means that xTestCluster is able to perfectly distinguish between correct and incorrect patches. For the remaining 21 bugs (56%), xTestCluster generates more than one cluster, and at least one of them includes correct and incorrect patches. This means that the generated test cases are not capable of detecting the "wrong" behavior of the incorrect patches, i.e., they are not capable of detecting, via new inputs, the differences between the behavior of correct and incorrect patches. Section 5.5 studies whether this limitation is related to the quality of the newly-generated tests.

Producing only pure clusters has two advantages. When selecting patches from a pure cluster, there is no risk of selecting an incorrect patch over a correct one. Moreover, if we know that a given cluster only contains correct patches, we can then safely select a patch from a cluster that satisfies perhaps an additional criterion, such as readability or patch length.

Answer to RQ3. Based on the generated test cases, xTestCluster is able to cluster plausible patches in a way that a single cluster will contain only correct or incorrect patches for 44% of the bugs considered (17/38). This signifies that for these bugs, it would be sufficient to assess just one of the patches from each cluster in order to check whether it contains a correct patch. Furthermore, a code reviewer can choose any patch from a cluster containing correct patches based on additional criteria that best fit their codebase.

5.3.2 Analysis of single cluster bugs

We previously focused on bugs with multiple clusters and the correctness of the patches that are contained in them. Now, we focus on bugs for which xTestCluster could not generate more than one cluster. As Table 3 shows, there are 51 of such bugs. For 41 of them (78.4%),

Table 4 RQ3. Classification of bugs based on cluster purity

Purity of clusters	#Bugs
Only pure clusters (either correct and incorrect)	17
At least 1 mixed cluster	21
Only pure incorrect (all patches are incorrect)	40
Only pure correct (all patches are correct)	0
Total (bugs with multiple clusters)	78

all patches are incorrect. This means that all incorrect patches produce the same behavior in the buggy program according to the test. The other 11 bugs (21.6%) contain correct and incorrect patches. However, xTestCluster cannot find any behavioral differences using generated test cases, which result is ‘passing’ (i.e., there is no error or failure assertion) on all patches. We will study such bugs with a single cluster in Section 5.5 to know whether this limitation is related to the quality of the newly-generated tests.

5.3.3 Relationships between overfitting patches and failing test cases

We now focus on the behaviors of patches from mixed clusters on the generated tests. As we previously mentioned, 21 bugs have one mixed cluster (and one or more pure clusters) and 10 bugs have a single mixed cluster. On the one hand, from the 21 bugs with multiple clusters, we found the patch from the mixed clusters fails on one or more newly-generated test cases. We recall that those failing test cases are provided from patches that belong to a different cluster. On the other hand, the remaining two bugs have a mixed cluster for which its patches do not fail any newly-generated test cases. With respect to the 10 bugs with a single mixed cluster, by construction there is no failing test.

We conclude that in the bugs with multiples clusters, one or them mixed, the behavior of the patches is diverse, and this diversity is *partially* detected by the newly-generated test cases, which are not fully capable of capturing some of the incorrect behaviour from overfitting patches.

5.4 RQ4: complementing state-of-the art patch assessment techniques

We now study to what extent our approach is able to complement two state-of-the-art patch assessment tools: Cache (Lin et al. 2022) and ODS (Ye et al. 2021).

We first focus on bugs for which xTestCluster generates pure clusters. We recall those bugs contain one or more clusters with only correct patches and one or more with only incorrect patches, but any cluster with both correct and incorrect. As discussed in Section 5.3.1, those bugs are 17.

After analyzing the assessment done by ODS and Cache on patches from those bugs, we observe that for 15 of those bugs, one of the assessment tools misclassify the correctness of a patch. In particular, ODS and Cache misclassify patches from 12 bugs and 8 bugs, respectively (three bugs are misclassified by both tools). For example, for bug Math-32 from Defects4J, xTestCluster groups patches in two ‘pure’ clusters: one with three incorrect patches, the other with one correct patch. ODS incorrectly classifies as overfitting the correct patch, and marks as correct one of the overfitting patches. In this case, xTestCluster provides developers ‘hints’ about that misclassification, in particular, let developers know that three patches (two truly incorrect but one wrongly classified as correct) behave similarly according to generated tests (that is, it provides a test case where all the three patches fail one assertion).

Answer to RQ4. Our results show that xTestCluster can complement the overfitting assessment analysis done by state-of-the-art assessment tool: combining patch assessment with cluster classification can expose misclassifications.

We also inspect the performance of ODS and Cache on the 21 bugs with multiple clusters and at least one mixed cluster. For each mixed cluster, we verify whether ODS and Cache are able to successfully recognize the overfitting patches from the correct ones. We recall that all these patches from a mixed cluster have the same behavior according to the tests generated from xTestCluster. Interestingly, for 17 of those bugs (81%), ODS or Cache incorrectly assess at least one of the patches included in a mixed cluster (in particular, ODS does that incorrect assessment for 16 bugs, Cache for 13, and both for 10 bugs).

Finally, we focus on the results of the assessment from ODS and Cache on the patches from bugs for which xTestCluster creates just a single mixed cluster, i.e., the cluster contains both correct and incorrect patches. As mentioned in Section 5.3.3 there are ten of such bugs. (The other 41 bugs with a single cluster contain only incorrect patches).

We find that for each of these 10 bugs, ODS or Cache produce at least one incorrect patch assessment: ODS fails on at least one patch assessment on nine bugs, and Cache does it on seven bugs. For example, the dataset from Wang et. al has five plausible patches from Closure-86 bug, all generated by SequenceR. According to the ground-truth provided by the dataset, only one of them is correct while the other four are incorrect. Ideally, xTestCluster would have had to generate *pure* clusters, one with the correct patch, and one or more clusters with the incorrect patches. However, both ODS and Cache could not identify any patch as correct: they wrongly classified all SequenceR patches as overfitting. This classification is a false positive.

ODS and Cache also suffer from false negatives. For example, there are six plausible patches from bug Math-59, one of them is overfitting (that one generated by SequenceR), while the other five are correct. xTestCluster groups them into a single cluster because it cannot observe behavioral differences between them. ODS and Cache cannot observe any differences: they classify all patches as correct. This produces a false negative on the classification of the overfitting patch.

Answer to RQ4 (cont.) Our results show that in most of the cases that xTest-Cluster cannot generate test that difference correct patches from overfitting, two state-of-art patch assessment tools produce false positive and/or negatives. This result shows the difficulty of patch assessment and calls for further research.

5.5 RQ5: quality of generated test cases and effectiveness of xTestCluster

To respond to RQ5, we first focus on the metrics computed in all test cases, then focus on the relationship between the metrics and the effectiveness of xTestCluster.

5.5.1 Quality metrics from test cases

Table 5 shows the mean, median, and standard deviation of four metrics computed and returned (together with the generated test cases) by Evosuite: *1)* Number of test cases generated for a given patch, *2)* lines of code (LOC) of such test cases, *3)* the line coverage of the generated test cases, and *4)* the mutation score. In this section, we focus exclusively on Evosuite for two main reasons: first, as we discuss in Section 6 Evosuite outperforms Randoop, secondly, the metrics are automatically generated by Evosuite.[7]

[7] Collecting data with EvoSuite https://www.evosuite.org/documentation/tutorial-part-3/

Table 5 Metrics from the generated test cases for all the patches considered

Metric	Mean	Median	Std Dev
# Tests per patch	98.45	46	114.12
LOC of a generated test cases for a patch	418.99	184	961.08
Line coverage	70%	77%	20%
Mutation Score	31%	21%	26%

Number of Test cases generated per patch[8]: We observe that the median value of generated test cases is 46 test. The average is larger (98.45) because, as we can see in Fig. 5c, there is still a considerable number of patches with between 200 and 400 generated tests.

Number of lines of code in the test cases[9]: We observe that the median value is 184 lines of code, which includes all the code from test cases generated for a patch. The average is also higher (418.45) because, as we can see in Fig. 5a, for some patches, there are outliers with more than 2000 test cases generated. Figure 5b shows the distribution without those outliers. We observe that most of the test generated have less than 1000 LOC.

Coverage: The coverage reported by Evosuite reaches a median coverage of 77% but the mean is just lower (70%). The distribution presented in Fig. 5d shows that most of the generated tests have a high coverage, between 80% and 100%, but still a considerable number of tests reach coverage between 40% and 0.6%.

Mutation Score: The mutation score reported by Evosuite reaches a mean of 31% and a lower median (21%). This means that most of the test cases generated are able to kill at most 21% of the generated mutants, which is notably low. The distribution presented in Fig. 5d seems to be a bimodal distribution: Most values are concentrated near 10% and near 55%. For very few patches, the generated tests reach a mutation score larger than 80%.

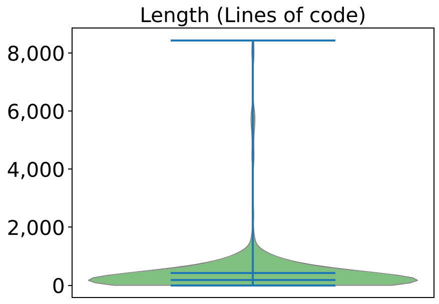
5.5.2 Relation between quality metrics and effectiveness of xTestCluster

Table 6 shows the median of each metric, but, as a difference from the previous section, we now consider a subset of tests. In particular, in column "Mixed Clusters" (second column), we consider test cases generated related to bugs with mixed clusters. In column "Pure Cluster" (third column), we consider test cases generated related to bugs with only pure clusters. The column "P-value" (fourth column) shows the p-value returned by the Mann-Whitney U test. Finally, the last column "Reject H_0" shows if the Null Hypothesis, which states that there is no significant difference between tests from mixed clusters and tests from pure clusters w.r.t. a metric, is rejected.

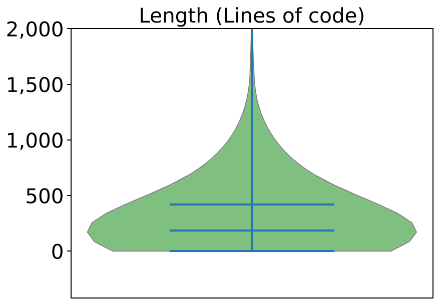
We observe that there are significant differences between the median of the number of test, LOC and line coverage, and using the Mann-Whitney U test we reject the null hypothesis for all these three metrics. For example, the patches assigned to mixed clusters have, as a median, 35 test cases. However, the patches assigned to pure clusters have, as a median, 126 test cases.

[8] In Evosuite this metrics is reported as "Size".

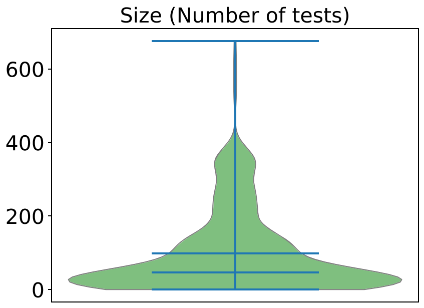
[9] In Evosuite this metrics is reported as "Length".



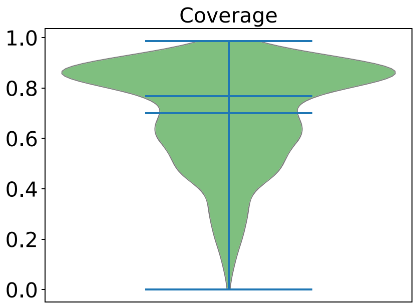
(a) Lines of code from test file



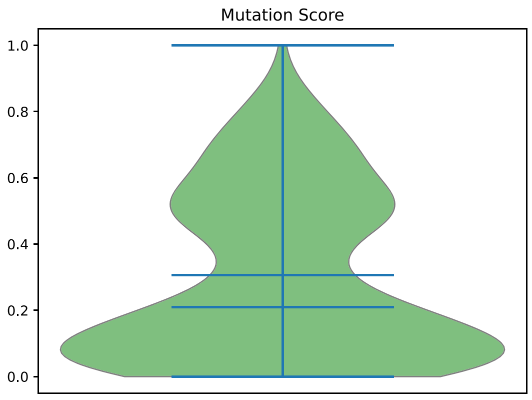
(b) Lines of code from test file (without outliers with length $>$ 2000 LOC)



(c) Number of test cases per test file



(d) Line coverage



(e) Mutation Score

Fig. 5 Metrics computed from newly-generated tests

Answer RQ5: Our results show that the effectiveness of xTestCluster is related to the quality of the test cases. Having a larger number of test cases, larger in terms of LOC and with higher coverage, help xTestCluster to detect behavioral differences between the correct and incorrect patches.

Table 6 Metrics computed on tests from patches assigned by xTestCluster to Mixed Clusters, and from those assigned to Pure Clusters

Metric	Mixed Cluster	Pure Cluster	P-value	Reject H_0?
# Tests per patch	35.00	126.00	<0.0001	True
LOC tests	142.50	392.00	<0.0001	True
Line coverage	74%	83%	0.0039	True
Mutation Score	19%	28%	0.1227	False

6 Discussion

In this section, we provide a deeper analysis of our findings, presenting two case studies, and an assessment of the performance of the two test generation tools we used in xTestCluster.

Case study: chart-26 with all pure clusters We collected 11 labelled patches for the bug Chart-26. After running the generated test cases on these patches, xTestCluster created three clusters as follows: xTestCluster first creates a cluster that has exclusively correct patches. Even though the patches are syntactically different, they have the same semantic behaviour. For example, a patch from JAID (Chen et al. 2017) adds an `if-return` in `Axis.java` file, whereas the patches from TBar (Liu et al. 2019) add an `if` guard to the same file. Then, xTestCluster creates two more clusters, both containing only syntactically different incorrect patches. In one cluster, two patches from JAID also affect the `Axis.java` file but introduce incorrect changes, such as variable assignment. However, in the other cluster, all incorrect patches affect a different file (i.e., `CategoryPlot.java`). Overall, xTestCluster not only clustered the correct patches together, but created clusters that contain patches that are semantically similar, despite having syntactic differences.

Performance of test-case generation tools Our implementation of xTestCluster has a parameter for choosing the test case generation tool to use. The parameter has three values: *1) EvoSuite* (Fraser and Arcuri 2011), *2) Randoop* (Pacheco and Ernst 2007), *3) both* (i.e., it uses the two tools). In this paper, we report the results using only Evosuite for different reasons. First, xTestCluster using Evosuite achieves better performance than using only Randoop (11 pure clusters). Second, the use of both tools (i.e., the addition of Randoop to our experimental setup) minimally impacts the performance (just one more bug with pure clusters -Closure 33-) but doubles the execution time. In conclusion, the addition of new test case generation tools can help xTestCluster to find more pure clusters at the expense of execution time.

Runtime overhead of xTestCluster We configure test generation tools with a timeout of one minute. The execution of the generated test cases takes a few seconds on average. Consequently, the overhead mostly depends on the number of generated patches that xTestCluster aims to cluster.

Reduction of execution time of xTestCluster The execution time (median 4.06 min per bug) can be reduced by parallelizing the generation of test cases for each patch, rather than sequentially executing them, as we do in this paper (loop in line 7 in Algorithm 2). The execution of these generated test cases can also be parallelized (loop in line 6 in Algorithm 3).

Selection of patches from a cluster All patches grouped in a cluster are semantically equivalent (i.e., behave similarly) according to both the human-written test cases and the new test cases generated by xTestCluster. Even the approach could randomly pick one of them as the representative of a cluster for the reason mentioned above, it could apply other selection strategies that go beyond the semantics of the patches: i.e., to consider the syntax of patches. For example, xTestCluster could apply a strategy based on the number of lines of source code it adds and removes from the original code, and favor shorter patches, i.e. those that result in least changed lines in the original code. The preference for shorter patches has also been implemented in previous work (e.g., Tian et al. (2022)), and has the advantage of helping the understandability and maintainability of the patched code (Fry et al. 2012).

Integration with existing APR xTestCluster can be easily integrated with any repair tool that generates Java patches, as it only requires the programs to be repaired and the generated patches as input. In this paper, our tool was used with patches generated by 25 repair tools.

Parameter fine-tuning In the experiment presented in this paper, xTestCluster invokes the test generation tools (Evosuite and Randoop) using the *default* values of each parameter, including the execution time (aka, *search budget*), which was set to one minute. Previous work investigated the effect of parameter optimization on test generation using EvoSuite (Arcuri and Fraser 2013). The authors conclude that using default values is a reasonable and justified choice, whereas parameter tuning (on test generation) is a long and expensive process that might or might not pay off in the end. Nevertheless, the application of parameter tuning could eventually improve the results presented in this paper. For example, increasing the execution time beyond the default time budget of 60 seconds provides Evosuite more time to find better quality test cases (in the context of this work, better test quality means, for instance, to strengthen the coverage on lines that have been patched). Beyond time, there are other parameters that could be fine-tuned with the goal of searching for improving the capacity of xTestCluster, such as the crossover rate, population size, elitism rate, selection, and parent replacement check. The impact of these five parameters on test generation was previously studied by Arcuri and Fraser (2013).

Limitations xTestCluster uses test generation tools to generate test cases for a patched program version. Consequently, the efficacy of xTestCluster heavily depends on the ability of these generation tools to create test cases (that is, a set of inputs and assertions of system output) that exercise buggy behavior affected by a patch. For example, if a generation tool generates test cases that do not cover a buggy line, then xTestCluster will not be able to differentiate the behavior of the patches and thus will create a single cluster.

7 Threats to validity

Next, we discuss potential issues regarding the implementation of xTestCluster (*internal validity*), the design of our study (*construct validity*), and the generalisability of our findings (*external validity*).

Internal validity The source code of xTestCluster and the scripts written for generating and processing the results of our experiments may contain bugs. This issue could have introduced a bias to our results, by removing or augmenting values. To mitigate this issue we made our source code, as well as the scripts used for the analysis and processing of the results of our study, publicly available in our repository (xtestcluster appendix repository 2021) for external validation.

Construct validity There is randomness associated with use of test-case generation tools such as EvoSuite (Fraser and Arcuri 2011) and Randoop (Pacheco and Ernst 2007), because of the nature of the algorithms used in such tools. Therefore, the results of our experiments could vary between different executions. In this experiment, we execute Randoop and EvoSuite once on each patch. Doing more executions of those tools (using different random seeds) could produce more diverse test cases that may find further differences between patches and, consequently, help xTestCluster produce better results.

External Validity Our study uses the dataset from Wang et al., which is in turn composed of two other datasets: DRR (Ye et al. 2021) (automated evaluation, using automated test cases generated on the human-patched program, taking it as ground truth) and the one from Liu et al. (2020) (manual evaluation done by humans). These two manners of labeling patches may affect our results. However, the patches from those work were then re-assessed for Wang et al, giving as a results the dataset with 902 patches that we use in this experiment.

Durieux et al. (2019) observed that the Defects4J benchmark might suffer from overfitting, as it has been used for the evaluation of most APR tools available. Thereby it can produce misleading results regarding the capabilities of APR tools to generate correct patches. However, since Defects4J has been used for the evaluation of most APR tools, we were able to find labeled data from multiple different APR tools only for Defects4J. Further research on the impact of using other bug benchmarks would tackle this threat.

8 Related work

In this section we discuss the most relevant related work and compare and contrast them to our proposed xTestCluster.

8.1 Patch clustering

Similar to xTestCluster, Cashin et al. (2019) present PATCHPART, a clustering approach that clusters patches based on invariants generated from the existing test cases. Using Daikon (Ernst et al. 2007) to find dynamic invariants and evaluated on 12 bugs (5 from GenProg and 7 from Arja), PATCHPART reduces human effort by reducing the number of semantically distinct patches that must be considered by over 50%. Our approach, in contrast, is based on output of the execution of newly generated test cases, and is validated on a larger set of bugs (139 vs 12), tools (25 vs 2). A deeper comparison with PATCHPART is not possible, as the information about the bugs repaired, considered patches, clusters generated and the tool is not publicly available.

Mechtaev et al. (2018) provide an approach for clustering patches based on test equivalence. There are two main differences between their work and ours. First, the main technical difference is that our clustering approach exploits additional automatically generated test cases, which are created to generate inputs that enforce diverse behavior, while Mechtaev et al. use solely the existing test suite written by developers, thus is unable to detect differences not exposed by unseen inputs. Secondly, the goal of our approach also differs: to group patches generated by different (and diverse) repair tools after those are generated, while Mechtaev et al. group patches from a single repair tool as they incorporate their approach to the patch generation process from one tool. For that reason, our evaluation considers patches from 25 repair tools for Java, while Mechtaev et al. evaluated four repair tools for C.

8.2 Patch assessment

Wang et al. (2020) provide an overview and an empirical comparison of patch assessment approaches. One of the core findings of Wang et al. (2020) is that existing techniques are highly complementary to each other. These overfitting techniques can be used to complement our work. For instance, we can apply xTestCluster, which is based on the cross-execution of newly generated test cases, on a set of previously filtered patches using automated patch assessment, or to use a patch ranking technique on the clusters created by xTestCluster. We describe a selection of such approaches in this section. Here we divide work on automated patch assessment into two categories: approaches that focus on overfitting as: (1) *independent tools*, which can be used with different APR approaches; (2) *dependent tools*, which are incorporated into specific repair tools.

Independent approaches Opad is a dynamic approach, which filters out overfitted patches by generating new test cases using fuzzing. Initially, Opad was developed for C programs and evaluated on GenProg, Kali, and SPR (Yang et al. 2017). Recently, a Java version for Opad has been introduced by Wang et al. (2020). There are several tools that use patch similarity for patch overfitting assessment. For instance, Patch- sim (Xiong et al. 2018) and Test- sim (Xiong et al. 2018) have been developed for Java and evaluated on jGenProg, Nopol, jKali, ACS and HDRepair. Specifically, the approach generates new test inputs to enhance original test suites, and uses test execution trace and output similarity to determine patch correctness. ObjSim (Ghanbari 2020) employs a similar strategy and has been evaluated on patches generated by PraPR. The above approaches involve code instrumentation, which can be costly. Like many other approaches, they also provide patch ranking as an output. DiffTGen (Xin and Reiss 2017) identifies overfitted patches by generating new test inputs that uncover semantic differences between an original faulty program and a patched program. Their test cases are created from an oracle, and in the evaluation of DiffTGen, the authors use the human-written patches as correctness oracle. Unlike them, in our work we do not assume existence of an oracle because it is not available during the repair process. Our approach creates new test cases from generated patches (not from human-written patches) and analyze the behavioural differences between them (not between a candidate patch and a human-written patch). The goals of our technique xTestCluster and the mentioned techniques are different. Those aim to classify patches as overfitting (in order to remove them), our technique clusters patches according to their behavior. Consequently, even if both aim to reduce human effort, the final goals are different.

Other patch assessment approaches that use machine learning (ML) to label correct and incorrect patches have recently emerged. For instance, ODS is a novel patch overfitting assessment system for Java that leverages static analysis to compare a patched program and a buggy program, and a probabilistic model to classify patches as correct or not (Ye et al. 2021). ODS can be employed as a post-processing procedure to classify the patches generated by different APR systems. Tian et al. (2020) demonstrated the potential of embeddings to empower learning algorithms in reasoning about patch correctness: a machine learning predictor with the BERT transformer-based embeddings associated with logistic regression.

Tian et al. also propose BATS (Tian et al. 2022), an unsupervised learning-based approach to predict patch correctness by statically checking the similarity of generated patches against past correct patches. In addition to the patch similarity, BATS also considers the similarity between the failing test cases that expose the bugs fixed by generated patches and those failing test cases that expose the bugs repaired by past correct patches. Kang and Yoo (2022) propose an approach based on language models which prioritizes patches that generate natural code. Cache from Lin et al. (2022) presents a deep learning-based classifier to predict the correctness of the patch. The evaluation of Cache shows that it performs better than the previous mentioned approaches including ODS and PATCH- SIM (Xiong et al. 2018). Liang et al. (2021) present an interactive filtering approach to patch review, which filters out incorrect patches by asking questions to the developers. The authors implemented this approach in an Eclipse plugin tool called InPaFer. The main differences between these machine learning approaches and XTESTCLUSTER, are as follows: (1) we aim to cluster patches based on behavioural differences, while the aforementioned approaches try to detect overfitting patches (2) the ML-based approaches are static (do not execute the patches), while our approach performs dynamic analysis by executing the generated patches using newly generated test cases (3) we do not require existence of a dataset of previously fixed patches.

Dependent approaches This category includes APR tools that also implement patch overfitting assessment techniques. Most techniques are based on static analysis and relevant heuristics. For instance, S3 is an APR tool for the C programming language that uses syntax constraints for assessing patch overfitting (Le et al. 2017). SSFIX is an APR tool for Java that leverages syntax constraints for assessing patch overfitting (Xin and Reiss 2017). CAPGEN considers programs' ASTs and a context-aware approach for assessing patch overfitting (Wen et al. 2018). PROPHET is an APR tool for C programs that rank patch candidates in the order of likely correctness using a model trained from human-written patches. Other techniques apply dynamic strategies for filtering overfitting patches. For instance, FIX2FIT (Gao et al. 2019) is an APR tool for C programs that defines a fuzzing strategy that filters out patches that make the program crash under newly generated tests. Our approach can complement these tools: XTESTCLUSTER can receive as input the (filtered) patches from one or more of those APR tools, and present to the user only those that behave differently.

9 Conclusions

We have introduced XTESTCLUSTER that is able to reduce the amount of patches required to be reviewed. XTESTCLUSTER clusters semantically similar patches together by exclusively utilising automated test generation tools. In this paper, we evaluate it in the context of automated program repair. APR tools can generate multiple plausible patches, that are not necessarily correct. Moreover, different tools can fix different bugs. Therefore, we consider a scenario where multiple tools are used to generate plausible patches for later patch assessment.

We use 902 patches from previous work that were labeled as correct or not and evaluated XTESTCLUSTER using that set. We show that XTESTCLUSTER can indeed cluster syntactically different yet semantically similar patches together.

Results show that XTESTCLUSTER reduces the median number of patches that need to be assessed per bug by half. This can save significant amount of time for developers that have to review the multitude of patches generated by APR techniques. Moreover, we provide test cases that show the differences in behavior between different patches.

For 44% of the bugs considered, all clusters are pure, i.e., contain only correct or incorrect patches. Thus, code reviewer can select any patch from such a cluster to establish correctness of all patches in that cluster.

In future work we will study the feasibility of complementing our approach with other techniques, e.g., patch ranking, in order to help xTestCluster to select the most adequate patch from a given cluster.

Funding We would like to gratefully acknowledge support from: Ramony Cajal Fellowship no. RYC2021-031523-I, ERC Advanced Grant no. 741278, UKRI EPSRC grant no. EP/P023991/1, Australian Research Council grant no. DP210100041.

Declarations

Conflict of Interests The authors declare that they have no conflict of interests.

References

Gazzola L, Micucci D, Mariani L (2017) Automatic software repair: A survey. IEEE Trans Software Eng 45(1):34–67

Monperrus M (2018) Automatic software repair: a bibliography. ACM Computing Surveys (CSUR) 51(1):1–24

Le XBD, Thung F, Lo D, Le Goues C (2018) Overfitting in semantics-based automated program repair. Empir Softw Eng 23(5):3007–3033

Long F, Rinard M (2016) An analysis of the search spaces for generate and validate patch generation systems. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, pp 702–713

Martinez M, Durieux T, Sommerard R, Xuan J, Monperrus M (2017) Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. Empir Softw Eng 22(4):1936–1964

Smith EK, Barr ET, Le Goues C, Brun Y (2015) 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, pp 532–543

Qi Z, Long F, Achour S, Rinard M (2015) An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Proceedings of the 2015 international symposium on software testing and analysis, pp 24–36

Ye H, Martinez M, Monperrus M (2021) Automated patch assessment for program repair at scale. Emp Softw Eng 26(2):20. [Online]. Available: https://doi.org/10.1007/s10664-020-09920-w

Martinez M, Monperrus M (2018) Ultra-large repair search space with automatically mined templates: the cardumen mode of astor. In: International symposium on search based software engineering. Springer, pp 65–86

Long F, Rinard M (2016) Automatic patch generation by learning correct code. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, ser. POPL '16

Ye H, Gu J, Martinez M, Durieux T, Monperrus M (2021) Automated classification of overfitting patches with statically extracted code features. IEEE Trans Softw Eng:1–1

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

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

Mechtaev S, Gao X, Tan SH, Roychoudhury A (2018) Test-equivalence analysis for automatic patch generation. ACM Trans Softw Eng Methodol 27(4):15:1–15:37. [Online]. Available: https://doi.org/10.1145/3241980

Cashin P, Martinez C, Weimer W, Forrest S (2019) Understanding automatically-generated patches through symbolic invariant differences. In: Proceedings of the 34th IEEE/ACM international conference on automated software engineering, ser. ASE '19. IEEE Press, pp 411–414. [Online]. Available: https://doi.org/10.1109/ASE.2019.00046

Lin B, Wang S, Wen M, Mao X (2022) Context-aware code change embedding for better patch correctness assessment. ACM Trans Softw Eng Methodol 31(3). [Online]. Available: https://doi.org/10.1145/3505247

Just R, Jalali D, Ernst MD (2014) Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 international symposium on software testing and analysis, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, pp 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

Wang S, Wen M, Lin B, Wu H, Qin Y, Zou D, Mao X, Jin H (2020) Automated patch correctness assessment: How far are we? In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, pp 968–980

Fraser G, Arcuri A (2011) EvoSuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th european conference on foundations of software engineering, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, pp 416–419

Pacheco C, Ernst MD (2007) Randoop: feedback-directed random testing for Java. In: Companion to the 22nd ACM SIGPLAN conference on object-oriented programming systems and applications companion, ser. OOPSLA '07. New York, NY, USA: Association for Computing Machinery, pp 815–816

xtestcluster appendix repository (2021) [Online]. Available: https://github.com/UPCArtifacts/xTestCluster

Durieux T, Madeiral F, Martinez M, Abreu R (2019) Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In: Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. ACM, pp 302–313

Liu K, Wang S, Koyuncu A, Kim K, Bissyandé TF, Kim D, Wu P, Klein J, Mao X, Traon YL (2020) On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, pp 615–627. [Online]. Available: https://doi.org/10.1145/3377811.3380338

Xiong Y, Wang J, Yan R, Zhang J, Han S, Huang G, Zhang L (2017) Precise condition synthesis for program repair. In: Proceedings of the 39th international conference on software engineering, ser. ICSE '17. IEEE Press, pp 416–426

Liu K, Koyuncu A, Kim D, Bissyandé TF (2018) AVATAR: fixing semantic bugs with fix patterns of static analysis violations. In: IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 1–12

Yuan Y, Banzhaf W (2018) ARJA: automated repair of Java programs via multi-objective genetic programming. IEEE Trans Softw Eng:1–1

Wen M, Chen J, Wu R, Hao D, Cheung S-C (2018) Context-aware patch generation for better automated program repair. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp 1–11

Martinez M, Monperrus M (2016) ASTOR: a program repair library for Java (demo). In: Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA). ACM, pp 441–444

Durieux T, Monperrus M (2016) DynaMoth: dynamic code synthesis for automatic program repair. In: 2016 IEEE/ACM 11th International Workshop in Automation of Software Test (AST), pp 85–91

Koyuncu A, Liu K, Bissyandé TF, Kim D, Klein J, Monperrus M, Le Traon Y (2020) FixMiner: mining relevant fix patterns for automated program repair. Emp Softw Eng:1–45

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

Liu K, Koyuncu A, Bissyandé TF, Kim D, Klein J, Le Traon Y (2019) You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp 102–113

Xuan J, Martinez M, DeMarco F, Clément M, Marcote SL, Durieux T, Berre DL, M (2017) Nopol: automatic repair of conditional statement bugs in Java programs. IEEE Trans Softw Eng 43(1):34–55

Liu X, Zhong H (2018) Mining stackoverflow for program repair. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 118–129

Chen Z, Kommrusch SJ, Tufano M, Pouchet L, Poshyvanyk D, Monperrus M (2019) SEQUENCER: sequence-to-sequence learning for end-to-end program repair. IEEE Trans Softw Eng:1–1

Jiang J, Xiong Y, Zhang H, Gao Q, Chen X (2018) Shaping program repair space with existing patches and similar code. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, pp 298–309

Hua J, Zhang M, Wang K, Khurshid S (2018) SketchFix: a tool for automated program repair approach using lazy candidate generation. In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, pp 88–891

Liu K, Koyuncu A, Kim D, Bissyandé TF (2019) TBar: revisiting template-based automated program repair. In: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, ser. ISSTA. ACM, pp 31–42

Le X-BD, Chu D-H, Lo D, Le Goues C, Visser W (2017) S3: syntax- and semantic-guided repair synthesis via programming by examples. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, pp 593–604

Xin Q, Reiss SP (2017) Leveraging syntax-related code for automated program repair. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering, ser. ASE. IEEE Press, pp 660–670

Liu K, Li L, Koyuncu A, Kim D, Liu Z, Klein J, Bissyandé TF (2021) A critical review on the evaluation of automated program repair systems. J Syst Softw 171:110817. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302156

Tian H, Li Y, Pian Y, Kaboré AK, Liu K, Habib A, Klein J, Bissyandé TF (2022) Predicting patch correctness based on the similarity of failing test cases. ACM Trans Softw Eng Methodol, just Accepted. [Online]. Available: https://doi.org/10.1145/3511096

Fry ZP, Landau B, Weimer W (2012) A human study of patch maintainability. In: Proceedings of the 2012 international symposium on software testing and analysis, ser. ISSTA 2012. 0.5em minus New York, NY, USA: Association for Computing Machinery, pp 177–187. [Online]. Available: https://doi-org.recursos.biblioteca.upc.edu/10.1145/2338965.2336775

Arcuri A, Fraser G (2013) Parameter tuning or default values? an empirical investigation in search-based software engineering. Emp Softw Eng 18(3):594–623. [Online]. Available: https://doi.org/10.1007/s10664-013-9249-9

Ernst MD, Perkins JH, Guo PJ, McCamant S, Pacheco C, Tschantz MS, Xiao C (2007) The Daikon system for dynamic detection of likely invariants. Sci Comput Program 69(1):35–45, special issue on Experimental Software and Toolkits

Yang J, Zhikhartsev A, Liu Y, Tan L (2017) Better test cases for better automated program repair. In: Proceedings of 2017 11th joint meeting of the european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17), 11 pages

Ghanbari A (2020) Objsim: lightweight automatic patch prioritization via object similarity

Tian H, Liu K, Kaboré AK, Koyuncu A, Li L, Klein J, Bissyandé TF (2020) Evaluating representation learning of code changes for predicting patch correctness in program repair. In: Proceedings of the 35th IEEE/ACM international conference on automated software engineering. ACM

Kang S, Yoo S (2022) Language models can prioritize patches for practical program patching

Liang J, Ji R, Jiang J, Zhou S, Lou Y, Xiong Y, Huang G (2021) Interactive patch filtering as debugging aid. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 239–250

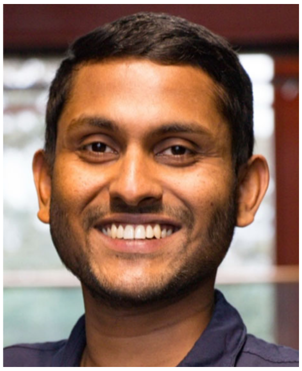
Wen M, Chen J, Wu R, Hao D, Cheung S-C (2018) Context-aware patch generation for better automated program repair. In: Proceedings of the 40th international conference on software engineering, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, pp 1–11. [Online]. Available: https://doi.org/10.1145/3180155.3180233

Gao X, Mechtaev S, Roychoudhury A (2019) Crash-avoiding program repair. In: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, ser. ISSTA 2019

Matias Martinez is a researcher from the Universitat Politècnica de Catalunya -BarcelonaTech- (Spain). He got his PhD degree from University of Lille (France) and a Computer Science degree from UNICEN (Argentina). He was previously an associate professor at the Université Polytechnique Hauts-de-France (France). His research focuses on empirical software engineering, automated software engineering, and software sustainability.

Maria Kechagia is a Research Fellow at University College London and a member of the Software Systems Engineering, CREST and SOLAR research groups. Previously, she was a postdoctoral researcher at the Delft University of Technology. She obtained a PhD degree in Software Engineering from the Athens University of Economics and Business and an MSc in Software Engineering from Imperial College London. Dr Kechagia has been an active and prolific researcher focusing her efforts on empirical software engineering, software verification (using static and dynamic analysis), and software optimisation (including GreenAI). On these topics, she has published high-quality scholarly articles in top-tier software engineering venues including TSE, EMSE, JSS, ICSE, and ISSTA.

She is a member of the Review Board of the Automated Software Engineering International Journal (Springer). She has been the co-chair of several software engineering events including the ISSTA'24 Demos track, SCAM'24 NIER track, COW65@UCL, APR' 22@ICSE, ISEC'22 Doctoral Symposium track, MSR'21 (Virtualisation chair), ICSME'21 (Publication chair), Euromicro DSD/SEAA STREAM track '20-'22. She has also served the programme committees of the research track of top-tier events (e.g., ICSE, FSE, ASE, ISSTA, MSR, and ICSME) and has been a regular reviewer for the major journals (e.g., TSE, TOSEM, EMSE, JSS, and AUSE).



Anjana Perera received the PhD degree from Monash University, Australia in 2022. He is currently a postdoctoral researcher in the Intelligent Application Security team at Oracle Labs, Australia. He was a software engineer, developing a latency critical, highly scalable and reliable electronic trading system for London Stock Exchange Group at LSEG Technology (formerly known as Millennium IT), until 2018. His research interests include automated test generation, search-based software engineering, and fairness testing of AI systems.

Justyna Petke is a Professor of Software Engineering at the Department of Computer Science, University College London. She is a member of Software Systems Engineering, CREST and SOLAR groups. Her current research focuses on Genetic Improvement. She also has expertise in Combinatorial Interaction Testing, Constraint Satisfaction and Search-Based Software Engineering. She held an EPSRC Early Career Fellowship on Automated Software Specialisation Using Genetic Improvement (2017-2023). Her work on applications of Genetic Improvement has been awarded 2 Humies (2014 2015), awarded for human-competitive results, and 2 distinguished paper awards (ISSTA 2015, ICSE-SEIP 2024).

She serves on the Editorial Board for the Journal of Systems and Software, Empirical Software Engineering, Genetic Programming and Evolvable Machines, Engineering Applications of Artificial Intelligence, and as Deputy Editor-in-Chief for the Automated Software Engineering journal.

Federica Sarro is a Professor of Software Engineering in the Department of Computer Science at UCL, where she is the Head of the Software Systems Engineering group and where she has established the SOLAR team within the CREST centre.

She has extensive academic and industrial expertise in Search-Based Software Engineering, Empirical Software Engineering and Software Analytics, with a focus on automated software management, optimisation, testing and repair. On these topics she has published over 100 peer-reviewed scholarly articles and given several invited talks at academic and industrial international events. She has also worked in collaboration with several companies including Meta, Google, JP Morgan Chase and Microsoft.

Professor Sarro has obtained numerous awards and generous funding for her research. In 2021, she was awarded the Rising Star Award by the IEEE Technical Community on Software Engineering in recognition of her “excellence in Software Engineering research with scholarly and real-world impact. Web page: https://profiles.ucl.ac.uk/38657-federica-sarro

Aldeida Aleti research focuses on addressing two significant challenges: enhancing the productivity of software developers through automation and AI, and ensuring the quality assurance of AI-based systems with a focus on responsible software engineering. Her contributions have resulted in over 85 publications in high-impact venues, including top conferences and journals in the field of Software Engineering. She has received numerous accolades for her research, including best paper awards and invitations to deliver keynote speeches at prestigious conferences. Currently serving as the Associate Professor and Associate Dean of Engagement & Impact in the Faculty of IT at Monash University, Aldeida has established herself as a prominent figure in academia and industry. Her research endeavors have garnered substantial funding, with over AUD 10M secured as Principal Chief Investigator (PCI) or Chief Investigator (CI) for various projects. Notable among these are grants from prestigious entities such as the Australian Research Council (ARC), CSIRO, Amazon AWS, and Facebook Research. In addition to her research achievements, Aldeida is deeply committed to nurturing the next generation of researchers. She has supervised 21 students at various levels, including PhD candidates, master’s students, and Honours students, guiding them to successful academic achievements. Her leadership extends beyond research supervision, as she has held formal roles such as Head of the Software Engineering Discipline group and Director of the Software Engineering Degree.

Authors and Affiliations

Matias Martinez[1] · Maria Kechagia[2] · Anjana Perera[3] · Justyna Petke[2] · Federica Sarro[2] · Aldeida Aleti[3]

✉ Federica Sarro
f.sarro@ucl.ac.uk

Matias Martinez
matias.martinez@upc.edu

Maria Kechagia
m.kechagia@ucl.ac.uk

Anjana Perera
Anjana.Perera@monash.edu

Justyna Petke
j.petke@ucl.ac.uk

Aldeida Aleti
aldeida.aleti@monash.edu

1 Universitat Politècnica de Catalunya, Barcelona, Spain

2 University College London, London, UK

3 Monash University, Clayton, Australia