Test-based Patch Clustering for Automatically-Generated Patches Assessment

Matias Martinez · Maria Kechagia ·
Anjana Perera · Justyna Petke ·
Federica Sarro · Aldeida Aleti

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 tools adoption in practice. Our work aims at increasing developer trust in automated patch generation by minimizing the number of plausible patches that they 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

Matias Martinez
Université Polytechique Hauts-de-France, France
E-mail: matias.martinez@uphf.fr

Maria Kechagia
University College London, United Kingdom
E-mail: m.kechagia@ucl.ac.uk

Anjana Perera
Monash University, Australia
E-mail: Anjana.Perera@monash.edu

Justyna Petke
University College London, United Kingdom
E-mail: j.petke@ucl.ac.uk

Federica Sarro
University College London, United Kingdom
E-mail: f.sarro@ucl.ac.uk

Aldeida Aleti
Monash University, Australia
E-mail: aldeida.aleti@monash.edu

more information about those patches for facilitating the 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 with 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 1910 plausible patches from 25 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 show the differences in behavior between generated patches.

1 Introduction

Automated program repair (APR) techniques generate patches for fixing software bugs automatically [1,2]. 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 [3,4,5]. 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 this test suite [6]. 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 [7].

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 [8], especially when multiple plausible patches are generated for a given bug [3,9]. 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. [10]), or as part of the post-processing of the generated patches (e.g. [11,12]). 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 [3] and as a result waste developers' time if the top-ranked patches are overfitted. Furthermore, such approaches might require an oracle [13], or (often expensive) program instrumentation [12], or a more sophisticated machine learning process [11].

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 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. [14] for patch generation and Cashin et al. [15] for patch assessment) that use solely the existing test suite written by developers. Our approach has the main advantage that it does not involve code instrumentation (aside from patch application) nor an oracle or pre-existing dataset to learn fix patterns. 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 to) 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. Finally, XTESTCLUSTER automatically selects one single patch from each cluster to show to the user.

In order to evaluate our approach we gathered recorded plausible patches from the literature that had been labeled as correct and/or overfitted (usually via manual effort). Overall, we gathered 1,910 plausible patches (679 correct and 1,231 overfitted) for bugs from v.1.5.0 of the DEFECTS4J data set [16], generated by 25 different APR tools. After removal of duplicates, we used two automated test-case generation tools, RANDOOP [17] and EVOSUITE [18], 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 patch. This is important because it shows that XTESTCLUSTER can be used in the wild, independently of the adopted Java repair tools.

Algorithm 1 xTestCluster

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

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 behavior for different patches generated for one bug. This additional information may help them decide which patch to select and merge into their codebase.

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. xTest-Cluster 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 [18] and Randoop [17], and a light-weight patch-length-based selection strategy. The code of xTestCluster is publicly available [19].
- An evaluation of xTestCluster using patches from 25 APR tools, and 1910 plausible patches.

All our data is available in our appendix [19].

2 Our approach

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

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 four steps (lines 3–6 of Algorithm 1): *1)* test-case generation, *2)* test-case execution, *3)* clustering, and *4)* patch selection.

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.
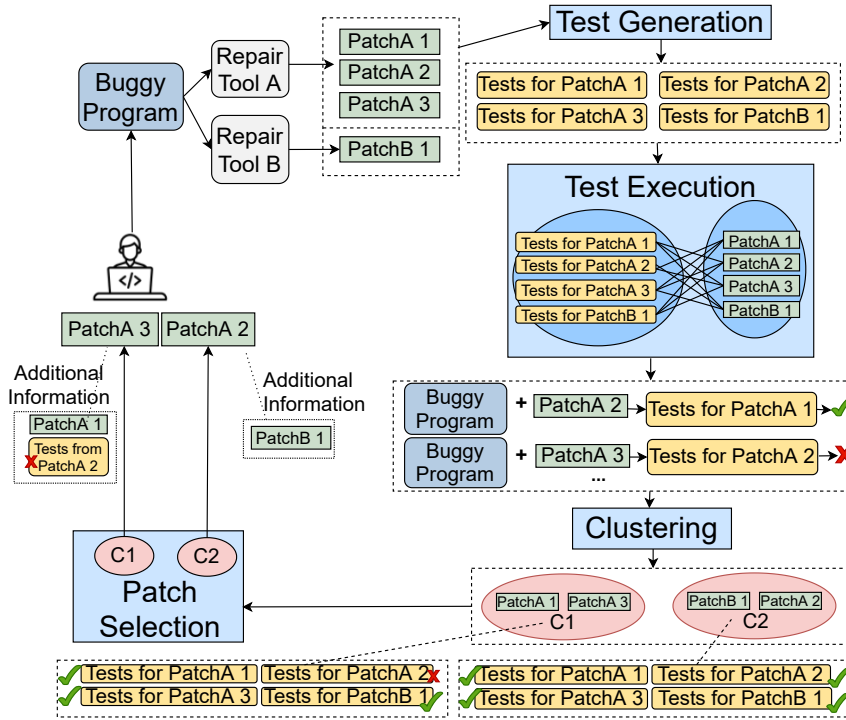
Fig. 1 The steps executed by xTestCluster.

Test-case execution. xTestCluster executes the generated test cases on each patched program version: 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 of a given buggy program. This cross execution aims to detect the behavioral differences among the generated patches that we use in clustering afterwards.

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.

Patch Selection. xTestCluster automatically selects one single patch for each cluster. This selected patch produces the same program behavior as all the other patches in a given cluster, with respect to our automatically generated test set. The selection is done based on a strategy, which is given as input to xTestCluster. This steps allows 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 instance, via failing test cases executed by a continuous integration platform (ci). Without using xTestCluster, the code reviewer needs to assess patches produced by repair

---

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

---

approaches that they have integrated, for instance, in their CI. Using XTEST-CLUSTER, the code reviewer can, now, review a *subset* from all the generated patches, reducing the review effort. Moreover, for each presented patch, they have also 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 next 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 the 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 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

---

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

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** $ResPatches_{exec}$

Algorithm 4 Clustering

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

$patch$, 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 5 Selection of Patches from Clusters

1: **Input:** Cs clusters, S selection heuristics
2: $selectedPatches \leftarrow \emptyset$
3: **for** $c \in Cs$ **do**
4:     $patches \leftarrow getPatchesFromCluster(c)$
5:     $selected \leftarrow selectPatch(patches, S)$
6:     $selectedPatches \leftarrow selectedPatches \cup selected$
7: **end for**
8: **return** $selectedPatches$

Algorithm 4 details this step. xTestCluster iterates over the patches (line 3) in order to assign each patch $patch_i$ to a cluster. If no cluster has been previously created (line 4), xTestCluster creates a new cluster that includes $patch_i$ (line 5). Otherwise, xTestCluster first retrieves the results from the previous, test case execution, step for that patch (line 7). Then, xTestCluster iterates over the created clusters (line 9). For each *cluster*, xTestCluster picks one patch $patch_o$ from it (line 10) and retrieves the corresponding results from the test case execution step (line 11). xTestCluster compares the two execution results (line 12): If both patches produce the same failures on our test set after they are applied to the buggy program, the patch $patch_i$ is included in *cluster* (line 13). Note that the result of a test case can be *passing* or *failing*. When the result is failing, we also consider the message associated 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 11 is empty) are cluster together. If xTestCluster cannot allocate $patch_i$ to any cluster (line 18), it creates a new cluster which includes $patch_i$ (line 19). Finally, xTestCluster returns all the created clusters (line 23).

2.4 Patch Selection

Algorithm 5 details the patch selection step. Given as input a set of clusters retrieved by the Clustering step (Algorithm 4), xTestCluster automatically selects one patch from each cluster. For each cluster (line 3), xTestCluster retrieves the corresponding patches (line 4), and based on a selection strategy S received as parameter, it selects one single patch (line 5). xTestCluster can accept any strategy that is able to select one patch from a list of patches. For example, it could apply simple strategies, e.g., random selection, based on heuristics (e.g., the length of a patch), or even delegate the selection to other approaches such as patch ranking (e.g., used by Prophet [10]). Finally, all selected patches are returned: Those are the patches which are presented to the code reviewer (line 8). At this point any of the current patch prioritisation techniques that were proposed in the literature for patch overfitting can be

employed. The question of which one is best in our context we leave for future work.

3 Research Questions

Our proposal, xTestCluster, aims at aiding code reviewers in reducing the effort required for the 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 at the Patch Selection stage would be sufficient to ensure existence (or non-existence) of a correct patch among *all* plausible patches in a cluster during patch assessment. This would make the runtime of Algorithm 5 be $O(c)$, where c is the number of clusters. In reality, we expect some clusters will have a mix of correct and incorrect patches, thereby motivating RQ4.

RQ4: Patch Selection Effectiveness *How often does selection of the shortest patch from a cluster return a correct patch?*

This RQs aims to measure the ability of a simple length-based patch selection strategy to select correct patches from a cluster that has both correct and incorrect patches. xTestCluster should prioritize correct over incorrect ones.

In this work we investigate if selection of the shortest patch would be sufficient. Intuitively, a short patch should also be easier to analyse by a code reviewer. This approach thus has also been used in APR tooling for prioritizing generated patches (e.g., in ARJA [20]). We also compare this approach with a random selection strategy. In theory any approach that tackles patch overfitting by prioritizing patches could be applied at this stage. We leave investigation of the best approach, that balances effectiveness vs. efficiency, for future work.

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., [8]) 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 [21]), 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 [16] dataset. To the best of our knowledge, DEFECTS4J is the most widely used dataset for the evaluation of repair approaches [5, 21, 22]. 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*.

Tools	Dataset of Patches					
	He et al. [8]		Liu et al. [22]		Kechagia et al. [23]	
	Correct	Overfitted	Correct	Overfitted	Correct	Overfitted
ACS [24]	18	5	33	10	-	-
Arja [25]	18	171	17	77	-	-
Avatar [26]	-	-	53	67	3	2
CapGen [27]	28	38	-	-	-	-
Cardumen [28]	0	0	5	23	1	1
DeepRepair [29]	5	9	-	-	-	-
DynaMoth [30]	-	-	4	31	1	3
Elixir [31]	26	15	-	-	-	-
FixMiner [32]	-	-	52	57	-	-
GenProg-A [25]	-	-	10	49	-	-
HDRepair [33]	6	3	-	-	-	-
Jaid [34]	42	39	-	-	-	-
JGenProg [28]	5	7	11	25	0	2
JKali [28]	0	0	8	25	0	2
JMutRepair [28]	0	0	10	23	0	2
Kali-A [25]	-	-	8	99	-	-
kPAR [35]	-	-	48	94	-	-
Nopol [36]	5	6	3	37	0	3
RSRepair-A [25]	-	-	13	62	-	-
Sequencer [37]	17	56	-	-	-	-
SimFix [38]	34	12	54	64	0	2
SketchFix [39]	16	9	-	-	-	-
SOFix [40]	22	2	-	-	-	-
ssFix [41]	15	9	-	-	-	-
TBar [42]	-	-	84	89	4	1
Total	257	381	413	832	9	18

Table 1 Number of Correct and Overfitted patches for bugs from Defects4J [16] per repair tool and patch dataset.

We found 3 datasets that met our criteria: 1) DDR by He et al. [8], which contains patches from 14 repair systems. He et al. [8] classified patches using a technique called RGT, which generates new test cases using a ground-truth, human-written oracle patches; 2) one dataset by Liu et al. [22], which includes patches from 16 repair systems, and manually evaluated the correctness using guidelines presented by Liu et al. [22]; 3) APIRepBench by Kechagia et al. [23], which includes patches from 14 repair tools. The patches were manually assessed.[2]

Table 1 presents the number of patches per repair tool and dataset. In total, from the three datasets, we collect 1910 patches. DDR has 638 patches (257 correct, i.e., 40%)[3], the dataset by Liu et al. [22] has 1245 patches (413 correct, i.e., 33%), and APIRepBench [23] has 27 patches (9 correct, i.e., 33%). We observe that the patches were generated by 25 repair tools. Moreover, patches generated by 13 out of the 25 tools were only reported in a single dataset. For

---

[2] Kechagia et al. [23] evaluated repair tools using bugs from Defects4J and from other bug benchmarks. We exclusively focus on patches for Defects4J.

[3] The repository of DDR [8] contains extra 625 patches that were unassessed. Consequently, He et al. [8] discarded those patches from their study. Thus, we also discard them in this work.

Summary of Bugs	#Bugs
Total bugs from DEFECTS4J [16]	375
Total bugs with labelled patches	226
Bugs with one patch	51
Bugs with > distinct one patch	175
Bugs considered by xTESTCLUSTER	139

Table 2 Bug summary from DEFECTS4J and their respective patches collected from the three datasets.

example, patches generated by ELIXIR [31] were curated and assessed by He et al. [8] but not by Liu et al. [22]. Inversely, patches generated by FIXMINER [32] were curated and manually assessed by Liu et al. [22] but not by He et al. [8] This shows the importance of considering multiple sources of patches.

Table 2 presents the number of bugs which the patches from the three datasets were generated for. Overall, our dataset of patches contains at least one correct patch for 226 bugs out of the 375 bugs contained in the version 1.5.0 of DEFECTS4J.

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.

From the patches gathered for 226 bugs in our dataset, we find that for 51 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 further 36 bugs for the following reasons: Firstly, patches for some bugs do not pass the existing program's test suites, i.e., are not actually plausible. For instance, for bugs Math-43, Lang-47 and Closure-3 all collected patches are not plausible. For further 11 bugs (e.g., Math-60, Closure-133, Closure-10) all patches except for one are not plausible. As we previously mentioned, xTESTCLUSTER needs at least two plausible patches per bug, thus those bugs are not considered. Wang et al. [43] also reported this issue with the dataset from Liu et al. [22]. Secondly, we discard bugs and associated patches if no tests cases are generated by our automated test generation tools. For instance, Randoop did not produce test cases for the Closure-21 bug. Overall, we consider 226-51-36 = 139 bugs in the evaluation of xTESTCLUSTER.

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. We use two test-case generation tools: Evosuite [18] and Randoop [17]. 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 timeout of 60 seconds (test-case generation). 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. For each bug, we compute the reduction of patches to analyze per bug B as follows:

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

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*.

For each of the sets of patches per bug, we compute the ability of xTestCluster to generate only pure clusters. Additionally, we compute the number of bugs having only pure clusters divided by the total number of bugs analyzed. The larger the result of the computation, the better.

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.

From our dataset of patches for 139 bugs, patches for 32 bugs are all correct, while patches for 42 bugs are all incorrect, leaving 65 bugs for which we have a mix of correct and incorrect patches – these are the ones we use to answer RQ3. We apply xTestCluster to the patches for these 65 bugs and check their `purity`, i.e., we calculate the number of clusters having all correct or all incorrect patches, and divide by the number of all clusters for a given bug.

4.5 RQ4: Patch Selection Effectiveness

To answer RQ4, we focus on the clusters marked as not pure in RQ3, as these clusters contain both correct and incorrect patches. The goal is to select correct

Bugs under Analysis	#Bugs
Bugs with one or more patches	226
Bugs with 1+ syntactically different patches	139
Bugs with multiple patches and clusters	68
Bugs with multiple patches in one single cluster	71

Table 3 RQ1. Generated clusters by xTestCluster per bug using EvoSuite and Randoop.

patches from them. To this end, we perform a preliminary analysis to measure the performance of two simple selection strategies: random and length-based selection.

We first conduct an experiment that randomly selects a patch from a cluster. For each cluster, we repeat this 100 times and report the average number of times that the selected patch is correct (according to the label provided in our dataset, see Section 4.1).

We also conduct an experiment that selects a patch based on the number of lines of source code it adds and removes from the original code. We favor shorter patches, i.e., those that result in least changed lines in the original code. Preference of shorter patches has been also implemented in previous work (e.g., [20]). This strategy can select multiple patches, i.e., they all can have equal length. In this case we report if all such patches are correct or not.
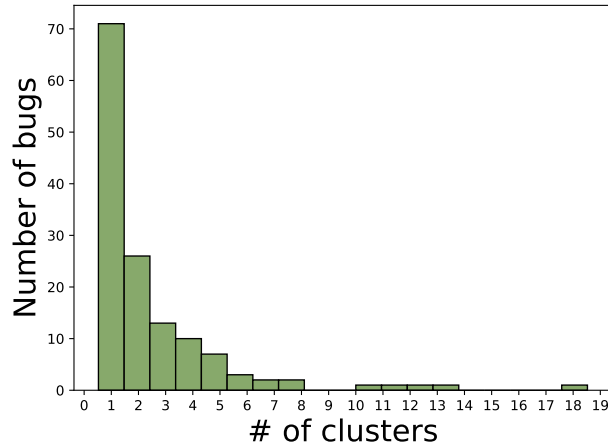
5 Results

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 68 bugs (49%) containing more than one plausible patch. This means that xTestCluster, using generated test cases, is able to differentiate between patches whose application produces different behavior.

For 71 bugs (51%), 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 syntactically 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, 139 bugs as explained in Section 4.2). We observe that the distribution is right-skewed. The most left bar corresponds to the previously mentioned 71 bugs with one cluster. Then, the number of

Fig. 2 RQ1. Distribution of the number of clusters per bug. Bugs with a single patch (in total 41) are discarded.

bugs with n clusters decreases as n increases. For 56 bugs, xTestCluster generated between two and five clusters, but for 12 bugs, it generates a larger number of clusters (six or more).
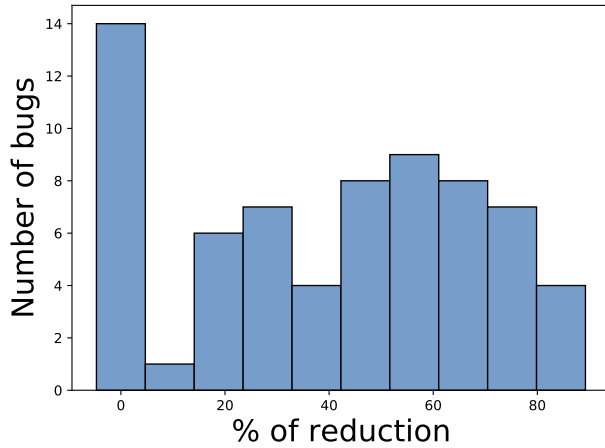
> **Answer to RQ1.** Given 139 bugs with at least two plausible and syntactically different patches, for 68 of them (49%), xTestCluster is able to detect patches with different behavior (based on test-case generation) and group them into different clusters.

By reviewing one patch per cluster, a code reviewer can reduce the time and effort required for reviewing, since they do not need to review all the generated patches for 49% of the bugs (68 in total). We also observe that those 68 bugs with multiple clusters have a median of eight patches (mean 10.5), while the other 71 bugs have a median of three (mean 4.7). This means that xTestCluster actually helps to greatly reduce the number of patches required to be reviewed.

5.2 RQ2: Patch Reduction Effectiveness

We report on the percentage reduction in the patches to be analyzed when xTestCluster is used vs. reviewing all available patches for a given bug. Figure 3 shows our results.

The median percentage reduction is 50% (mean 44.67%), which means that for half of the bugs (34) xTestCluster reduces the total number of patches one needs to analyze by at least 50%. For four bugs the reduction is even larger (80%).

Fig. 3 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.

The distribution in Figure 3 also shows that for 14 bugs we achieve no reduction. That is, for 14 bugs each cluster contains a single patch, thus all need to be analyzed. Nevertheless, for most of those cases (10), the number of patches (and clusters) is two. In other words, a code reviewer only needs to check two patches per bug. Moreover, our approach could help them decide which is better, as xTestCluster provides test cases on which the application of the two patches would produce a different behavior.

> **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.

5.3 RQ3: Clustering Effectiveness

We now focus on the 68 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.

Purity of Clusters	#Bugs
Only pure (either correct and incorrect)	15
At least 1 mixed cluster	24
Only pure correct	3
Only pure incorrect	26
Total (bugs with multiple clusters)	68

Table 4 RQ3. Classification of bugs based on cluster purity.

We first observe that 29 bugs have either only correct patches (three bugs) or only incorrect patches (26 bugs), thus all clusters generated for those bugs are pure by construction.

For 15 out of the remaining 39 bugs, all clusters generated by xTestCluster are pure. This means that xTestCluster is able to distinguish between correct and incorrect patches.

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 38% of bugs considered. 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. Moreover, a code reviewer can choose any patch from a cluster containing correct patches based on additional criteria that best fit their codebase.

For 24 bugs (62%) at least one mixed cluster was generated, that includes both correct and incorrect patches. This means that the generated test cases are not able to detect the "wrong" behavior of the incorrect patches. In these cases, xTestCluster needs to apply a patch selection strategy. This step is further investigated in RQ4.

5.4 RQ4: Patch Selection Effectiveness

Table 5 shows a summary of our results for RQ4. In total, from the results presented in RQ3 (Section 5.3), xTestCluster generated 27 mixed clusters for 24 bugs.

We use a length-based selection strategy and compare it with randomly sampling one patch per cluster (as described in Section 4.5).

	#Clusters
Total mixed clusters	27
Shortest selection strategy	
The shortest patches are all correct patches	13 (48%)
1+ correct and 1+ incorrect are the shortest	8 (30%)
Total shortest includes a correct patch	21 (78%)
Random selection strategy	
# Times correct is randomly selected (avg.)	13.5 (50%)

Table 5 RQ4. Evaluation of the random and length-based patch selection strategies.

Our results show that the random strategy selects a correct patch with 50% probability. In particular, from 100 sampling runs, a correct patch is selected for 13.5 out of 27 clusters on average.

The length-based strategy produces similar results. In total, for 21 mixed clusters (78%), at least one patch that causes least line changes to the original code (which we call *shortest*) is correct. For 13 of these 21 clusters, all the shortest patches are correct. In the other eight cases, there are multiple shortest patches not all of which are correct. In particular: in four clusters, at least 80% of such patches are correct; in one cluster 66% of the patches are correct; in two clusters half of the shortest are correct; and only in one cluster the shortest patches are mostly incorrect.

> **Answer to RQ4.** Our results show that the shortest patch selection strategy selects a correct patch in 50% of cases. This is comparable to random selection, thus other approaches should be investigated in the future.

We note that both strategies we tried are quite simple and quick to compute. Therefore, their results allow us to define a lower bound for future work. We intend to study other, more sophisticated, patch selection strategies, for instance, focused on ranking (e.g. [10]), or on source-code features and pattern-based learning (e.g., [11,43,44]) in future work. Such strategies can allow XTESTCLUSTER to not only focus on better selection of correct patches over incorrect ones, but also provide opportunity to select patches based on other criteria, such as patch readability.

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 1: Chart-26 with all pure clusters. We collected 17 labelled patches for the bug Chart-26. After running the generated test cases

on these patches, xTestCluster created three clusters as follows: xTest-Cluster first creates one cluster that has exclusively correct patches. Even though the patches are syntactically different, they have the same semantic behaviour. For example, a patch for JAID [34] adds an `if–return` in `Axis.java` file, whereas the patches from TBar [42] 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 a variable assignment. Whereas, in the other cluster, all the incorrect patches affect a different file (i.e., `CategoryPlot.java`). Overall, xTestCluster not only clustered correct patches together, but created clusters that contain patches that are semantically similar, despite having syntactic differences.

Case Study 2: Lang-35 with two pure clusters All the patches for Lang-35 were labeled as correct in previous work [8, 22]. There are two syntactically different patches that xTestCluster groups into two different clusters. One patch created by ACS [24], introduces an `if` condition with a `throw` statement, while the other patch created by Arja [25], replaces two statements by using a `throws` clause. A test case created for the latter patch fails when it is executed on the ACS patch. Since these the two patches behave differently, xTestCluster correctly places each in a separate cluster. We conclude that xTestCluster provides code reviewer with valuable information: It is able to flag whether the two patches are semantically different, and to provide them with the test inputs (encoded in the mentioned generated failing test) required to expose the difference in their behavior.

Performance of Test-Case Generation Tools. In our experiments, we set up xTestCluster to use two test-case generation tools, EvoSuite [18] and Randoop [17]. We carry out an additional experiment that executes xTestCluster using only one test-case generation tool at a time (either EvoSuite or Randoop). The goal of this experiment is to measure the ability of each tool to detect behavioral differences between correct and incorrect patches. In particular, we measure the number of bugs for which xTestCluster generates only pure clusters. As we show in Section 5.3, using both test-case generation tools, xTestCluster generates pure clusters only for 15 bugs. Whereas, xTestCluster using only Randoop finds pure clusters for 11 bugs, and xTestCluster using only EvoSuite finds pure clusters for 12 bugs. This means that there are patches for three bugs which xTestCluster with EvoSuite is able to detect behavioral differences for, but with Randoop alone it is not able to do so. Conversely, xTestCluster with Randoop detects differences between patches for three bugs, which are not detectable using only EvoSuite. These results are aligned with those presented by Shamshiri et al. [45] on measuring the ability of generated test cases to detect real-world bugs, i.e., some bugs are only detected by test cases generated by a single tool (either EvoSuite or Randoop). To conclude, for xTestCluster, EvoSuite and Randoop are complementary, and the use of both tools helps xTestCluster increase its performance.

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.

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.

7 Threats to Validity

Next, we discuss potential issues regarding the implementation of xTest-Cluster (*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 [19] for external validation.

Construct Validity. There is randomness associated with use of test-case generation tools such as EvoSuite [18] and Randoop [17], 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. The study combines three datasets, whose labels were produced in different ways, either by human or automated evaluation, and this may affect our results. In particular, the labels produced by Kechagia et al. [23] were found by human assessment (i.e., two validators independently assessed each patch), while the labels produced by Ye et al. [8] were found by using automated test cases generated on the human-patched program, taking it as ground truth. Additionally, Le et al. [46] found that automated testing performs worse than human labeling, whilst others have found human labeling introduces bias [8]. Thus, since both manual and automated assessment were used for labels we examined, there may be some quality deficiencies. However, we argue that the dataset we curated for this study is the largest and most comprehensive publicly available, having a greater number of test-cases generated for each bug in comparison with the one curated in previous work [46]. The magnitude of our dataset could mitigate such labeling issues. Further research should be done using other labeled sets once they become available.

Durieux et al. [21] 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 proposal xTESTCLUSTER.

8.1 Patch clustering

Similar to xTESTCLUSTER, Cashin et al. [15] present PATCHPART, a clustering approach that clusters patches based on invariants generated from the existing test cases. Using Daikon [47] 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 [14] 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. [43] provide an overview and an empirical comparison of patch assessment approaches. One of the core findings of Wang et al. [43] 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 [48]. Recently, a Java version for Opad has been introduced by Wang et al. [43]. There are several tools that use patch similarity for patch overfitting assessment. For instance, Patch-sim [12] and Test-sim [12] 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 [49] 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 [13] 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 a 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 [11]. ODS can be employed as a post-processing procedure to classify the patches generated by different APR systems. Tian et al. [44] 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 [50], 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 et al. [51]

propose an approach based on language models which prioritizes patches that generate natural code. 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 [52]. ssFix is an APR tool for Java that leverages syntax constraints for assessing patch overfitting [53]. CapGen considers programs' ASTs and a context-aware approach for assessing patch overfitting [54]. 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 [55] 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 gathered 1910 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 38% of bugs considered all the clusters are pure, i.e., contain only correct or incorrect patches. Thus, code reviewer can select any patch from such a cluster to establish correctes of all patches in that cluster. In the other

cases we need a selection strategy so that correct patches are selected before incorrect ones.

We thus assess the feasibility of using xTESTCLUSTER with two simple patch selection strategies (i.e., length-based and random), which provides a baseline for future investigation with other patch selection strategies. These can be existing patch overfitting techniques that provide patch ranking. In this way our approach is complementary to existing work on patch overfitting.

References

1. L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2017.
2. M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018.
3. X. B. D. Le, F. Thung, D. Lo, and C. Le Goues, "Overfitting in semantics-based automated program repair," *Empirical Software Engineering*, vol. 23, no. 5, pp. 3007–3033, 2018.
4. F. Long and M. Rinard, "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, 2016, pp. 702–713.
5. M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.
6. E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 532–543.
7. Z. Qi, F. Long, S. Achour, and M. Rinard, "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*, 2015, pp. 24–36.
8. H. Ye, M. Martinez, and M. Monperrus, "Automated patch assessment for program repair at scale," *Empirical Software Engineering*, vol. 26, no. 2, p. 20, 2021. [Online]. Available: https://doi.org/10.1007/s10664-020-09920-w
9. M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in *International Symposium on Search Based Software Engineering*. Springer, 2018, pp. 65–86.
10. F. Long and M. Rinard, "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, 2016.
11. H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, "Automated classification of overfitting patches with statically extracted code features," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.
12. Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 789–799.
13. Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 226–236.
14. S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury, "Test-equivalence analysis for automatic patch generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 4, pp. 15:1–15:37, 2018. [Online]. Available: https://doi.org/10.1145/3241980
15. P. Cashin, C. Martinez, W. Weimer, and S. Forrest, "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, 2019, p. 411–414. [Online]. Available: https://doi.org/10.1109/ASE.2019.00046

16. R. Just, D. Jalali, and M. D. Ernst, "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, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

17. C. Pacheco and M. D. Ernst, "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, 2007, pp. 815—-816.

18. G. Fraser and A. Arcuri, "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, 2011, pp. 416—-419.

19. "xtestcluster appendix repository." [Online]. Available: https://anonymous.4open.science/r/xTestCluster-1803/

20. Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *TSE*, vol. 46, no. 10, pp. 1040–1067, 2018.

21. T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "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, 2019, pp. 302–313.

22. K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "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, 2020, p. 615–627. [Online]. Available: https://doi.org/10.1145/3377811.3380338

23. M. Kechagia, S. Mechtaev, F. Sarro, and M. Harman, "Evaluating automatic program repair capabilities to repair api misuses," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

24. Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 416–426.

25. Y. Yuan and W. Banzhaf, "ARJA: Automated repair of Java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

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

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

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

29. M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 479–490.

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

31. R. K. Saha, H. Yoshida, M. R. Prasad, S. Tokumoto, K. Takayama, and I. Nanba, "Elixir: An automated repair tool for java programs," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE. ACM, 2018, pp. 77–80.

32. A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "FixMiner: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, pp. 1–45, 2020.

33. X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 213–224.

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

35. K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "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)*, 2019, pp. 102–113.

36. J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. L. Berre, and M., "Nopol: Automatic repair of conditional statement bugs in Java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

37. Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "SEQUENCER: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

38. J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "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, 2018, p. 298–309.

39. J. Hua, M. Zhang, K. Wang, and S. Khurshid, "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, 2018, p. 888–891.

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

41. Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 660–670.

42. K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA. ACM, 2019, pp. 31—-42.

43. S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "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, 2020, pp. 968—-980.

44. H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "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, 2020.

45. S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015, p. 201–211. [Online]. Available: https://doi.org/10.1109/ASE.2015.86

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

47. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007, special issue on Experimental Software and Toolkits.

48. J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *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*, 2017.

49. A. Ghanbari, "Objsim: Lightweight automatic patch prioritization via object similarity," 2020.

50. H. Tian, Y. Li, W. Pian, A. K. Kaboré, K. Liu, A. Habib, J. Klein, and T. F. Bissyandé, "Predicting patch correctness based on the similarity of failing test cases," *ACM Trans. Softw. Eng. Methodol.*, jan 2022, just Accepted. [Online]. Available: https://doi.org/10.1145/3511096

51. S. Kang and S. Yoo, "Language models can prioritize patches for practical program patching," 2022.

52. X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: Syntax- and semantic-guided repair synthesis via programming by 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, 2017, pp. 593—-604.

53. Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017.   IEEE Press, 2017, p. 660–670.

54. M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "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, 2018, p. 1–11. [Online]. Available: https://doi.org/10.1145/3180155.3180233

55. X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, 2019.