On the Efficiency of Test Suite based Program Repair
A Systematic Assessment of 16 Automated Repair Systems for Java Programs

Kui Liu∗
brucekuiliu@gmail.com
Nanjing University of Aeronautics and Astronautics
China

Shangwen Wang†
wangshangwen13@nudt.edu.cn
National University of Defense Technology
China

Anil Koyuncu
Kisub Kim
{anil.koyuncu,kisub.kim}@uni.lu
University of Luxembourg
Luxembourg

Tegawendé F. Bissyandé
tegawende.bissyande@uni.lu
University of Luxembourg
Luxembourg

Dongsun Kim
darksw@furiosa.ai
Furiosa.ai
Republic of Korea

Peng Wu
wupeng15@nudt.edu.cn
National University of Defense Technology
China

Jacques Klein
jacques.klein@uni.lu
University of Luxembourg
Luxembourg

Xiaoguang Mao
xgmao@nudt.edu.cn
National University of Defense Technology
China

Yves Le Traon
yves.letraon@uni.lu
University of Luxembourg
Luxembourg

ABSTRACT
Test-based automated program repair has been a prolific field of research in software engineering in the last decade. Many approaches have indeed been proposed, which leverage test suites as a weak, but affordable, approximation to program specifications. Although the literature regularly sets new records on the number of benchmark bugs that can be fixed, several studies increasingly raise concerns about the limitations and biases of state-of-the-art approaches. For example, the correctness of generated patches has been questioned in a number of studies, while other researchers pointed out that evaluation schemes may be misleading with respect to the processing of fault localization results. Nevertheless, there is little work addressing the efficiency of patch generation, with regard to the practicality of program repair. In this paper, we fill this gap in the literature, by providing an extensive review on the efficiency of test suite based program repair. Our objective is to assess the number of generated patch candidates, since this information is correlated to (1) the strategy to traverse the search space efficiently in order to select sensible repair attempts, (2) the strategy to minimize the test effort for identifying a plausible patch, (3) as well as the strategy to prioritize the generation of a correct patch. To that end, we perform a large-scale empirical study on the efficiency, in terms of quantity of generated patch candidates of the 16 open-source repair tools for Java programs. The experiments are carefully conducted under the same fault localization configurations to limit biases. Eventually, among other findings, we note that: (1) many irrelevant patch candidates are generated by changing wrong code locations; (2) however, if the search space is carefully triaged, fault localization noise has little impact on patch generation efficiency; (3) yet, current template-based repair systems, which are known to be most effective in fixing a large number of bugs, are actually least efficient as they tend to generate majoritarily irrelevant patch candidates.

CCS CONCEPTS
• Software and its engineering → Software verification and validation; Software defect analysis; Software testing and debugging.

KEYWORDS
Patch generation, Program repair, Efficiency, Empirical assessment.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE ’20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7121-6/20/05 . . . $15.00
https://doi.org/10.1145/3377811.3380338

1 INTRODUCTION
In the last decade, Automated Program Repair (APR) [11, 26, 41] has extensively grown as a prominent research topic in the software engineering community. Figure 1 overviews the research activities of this topic. The associated literature includes a broad
range of techniques that use heuristics (e.g., via random mutation operations [25]), constraints solving (e.g., via symbolic execution [44]), or machine learning (e.g., via building a code transformation model [13]) to drive patch generation. A living review of automated program repair research appears in [42], which shows that the research in this field has been revived with the seminal work, ten years ago, of Weimer et al. [56] on generate-and-validate approaches. Patches are generated to be applied on a buggy program until the patched program meets the desired behaviour. In the absence of formal specifications of the desired behaviour, test suites are leveraged as affordable partial specifications for validating generated patches. Over the years, the community has incrementally advanced the state-of-the-art with numerous test-based approaches that have been shown effective in generating valid patches for a significant fraction of defects within well-established benchmarks [16, 27, 36, 49].

Figure 1: APR research publications since 2009\(^1\).

Several studies have revisited the constraints and performance of program repair systems, and have thus contributed to shaping research directions towards improving the state-of-the-art. For example, Qi et al. [48] have early shown that repair tools generate mostly overfitting patches (i.e., patches that pass the incomplete test cases) but are actually incorrect. Their study led to assessment results being now carefully presented in a way that highlights the capability of new approaches to correctly repair programs. Motwani et al. [43] then questioned whether state-of-the-art approaches can deal with hard and important bugs. Liu et al. [29] recently revealed significant biases with fault localization configurations in APR system evaluations. More recently, Durieux et al. [7] have shown that state-of-the-art tools may actually be overfitting the associated study benchmarks.

Performance measurement of repair systems has evolved to progressively consider the number of correctly-fixed bugs or the diversity of benchmark bugs [7] that are fixed. Another performance aspect that deserves investigation is the efficiency of the patch generation system. It is however mentioned in only a few assessment reports [12, 63]. Yet, efficiency is a key property for bringing repair tools to specific bugs, efficiency can be substantially improved.

1 Data are extracted from Monperrus’s living review on APR [42].

source code). This criterion, which was already mentioned in a few of the previous work [33, 57, 63], however, has limitations with respect to generalizability (cf. Section 2): execution time is (1) dependent on many variables that are unrelated to the approach implemented in the repair system; and (2) is generally unstable.

We postulate that the efficiency of test-based program repair should be assessed along with the following question: how many attempts does the repair system make before catching a valid patch? In previous work, Qi et al. [47] have formulated this question into a metric that served to assess the effectiveness of fault localization techniques in a platform-agnostic manner. To the best of our knowledge, little attention has been paid to measuring repair efficiency by estimating the number of validated patch candidates.

In this paper, we report on the results of a large scale empirical study on the efficiency of test-based program repair systems. Our study considers 16 APR systems targeting Java programs, and performs a systematic assessment under identical and controlled fault localization configurations. The objective of this work is to contribute a comprehensive analysis of repair efficiency to the literature with respect to generated patches for a large spectrum of APR systems. Eventually, we gather insights on how the strategies of approaches in the literature affect repair efficiency. Overall, we mainly find that:

F0: So far, efficiency is not a widely-valued performance target.

We found that state-of-the-art APR tools are the least efficient. This calls for an industry investigation of the impact of efficiency on adoption (or lack thereof).

F1: Across time, repair tools subsume each other in terms of which benchmark bugs can be fixed. Unfortunately, effectiveness (i.e., how many bugs are eventually fixed) is increased at the expense of efficiency (i.e., how many repair attempts are made before a given bug is fixed).

F2: Template-based repair systems are generally inefficient as they produce too many patch candidates. However, when the templates are mined from clean datasets or are specialized to specific bugs, efficiency can be substantially improved.

F3: Literature approaches develop a few strategies, such as constraint solving or donor code search, which contribute to drastically reducing the nonsensical or in-plausible patches.

F4: APR systems that implement random search over the repair search space require large sets of patch candidates to increase the likelihood of hitting a correct patch.

F5: Implementation details can diversely influence the repair efficiency of an APR approach.

2 BACKGROUND AND MOTIVATION

Test suite based program repair systems commonly implement a three-step pipeline as illustrated in Figure 2: fault localization, which produces a ranked list of suspicious code locations that should be modified to fix the bug; patch generation, which implements the change operators that are applied on the buggy code locations; and patch validation, which executes the test cases to check that the patched program meets the behaviour (approximately) specified by the test suite.

If a patch candidate can pass all the given test cases (both previously-passing and previously-failing test cases on the buggy version), it is
This section presents the design details of this empirical study.

3.1 Research Questions

Overall, our investigation into the efficiency of test-based APR systems seeks answers for the following research questions (RQs):

(1) **RQ1. Repairability across time**: We first revisit the classical performance criterion of APR systems, which is about the repairability (i.e., effectiveness): *how many bugs can be fixed by test suite based repair approaches?* Our investigation goes beyond previous studies in the literature by (i) systematically assessing a large range of repair systems under the same configurations (see Section 3.3.2); and (ii) exploring not only plausibility but also the correctness of patches (see Section 3.3.3). Eventually, we investigate the evolution across time of effectiveness to better discuss the need for revisiting efficiency as an important complementary performance criterion.

(2) **RQ2. Patch generation efficiency**: Based on the experimental outputs of benchmarking repair systems in RQ1, we can investigate the efficiency of test-based repair: *how many patch candidates are generated and checked before fixing a given bug?* Although program repair is often regarded as a background/online task, efficiency remains critical since resource budgets are limited. Therefore, efficiency may have adverse effects on the adoption of the repair system and even on its effectiveness. In this RQ, we extensively review two cases of invalid patches.
whose generation may undermine efficiency: nonsensical and in-plausible patches (see Section 3.5).

(3) \textbf{RQ3. Fault Localization noise impact on efficiency:} Finally, given that fault localization is known to provide noisy inputs to repair, we investigate its impact on efficiency to highlight repair directions for mitigations. Mainly, we question whether some repair strategies are more or less resilient to repair attempts on wrong code locations. Our study differs from recent work [29] in the literature, which explores the bias of fault localization on repairability with only one repair system.

### 3.2 Subject Selection

Our study focuses on APR systems targeting Java programs. Java is indeed today the most targeted language in the community of program repair. Furthermore, a well-formed dataset of real-world Java program bugs is available, with the necessary tool support to readily compile and execute programs. Although we initially planned to consider all repair approaches proposed in the last decade, we were limited by the fact that many APR tools are not open-source or even publicly available.

In the end, APR systems considered for our study are systematically selected based on the following criteria:

1. \textit{Availability}: our study involves the execution of APR tools, thus APR approaches without publicly available tools are excluded.
2. \textit{Executability}: some APR approaches provide publicly available tools, which however cannot be executed as-is for diverse issues (e.g., ssFix [61] failed to execute because of an online connection to a private search engine fails). We exclude such approaches from the study.
3. \textit{Configurability}: to limit biases, we need to configure the different tools to use the same input information (e.g., fault localization details). We, therefore, exclude APR approaches whose tools cannot be readily configured. For example, HDRepair [22] implementation is tied to an assumption that exact information on the faulty method is first available.
4. \textit{Standalone}: finally, our selection ensures that we focus on APR approaches where the tools can be run if provided with Java program source code and the available test suite. Therefore, any tool that would require extra data is excluded (e.g., LSRRepair [32] requires run-time code search over Github repositories).

We consider two sources of information to identify Java APR tools: the community-led program-repair.org website and the living review of APR by Monperrus [42]. As of July 2019, 31 APR tools were targeting Java programs listed in the literature. After systematically examining these tools, 16 are found to satisfy our criteria and are therefore finally selected. Table 2 enumerates all Java-based APR tools and provides arguments for rejection/consideration. We categorize them into three main categories: heuristic-based [26], constraint-based [26], and template-based [17] repair approaches.

\begin{table}[h]
\caption{Selected and excluded APR tools for our study.}
\begin{tabular}{|c|c|c|}
\hline
\textbf{Selected} & \textbf{Reason} & \textbf{APR Tools for Java Programs} \\
\hline
\textbf{No} & Not public & PAR [17], zPAR [22], jFix/S5 [25], ELIDIR [20], Hermes [11], SOFIM [10], CapGem [27], FixPR [10] \\
\hline
\textbf{No} & Faulty method required & HDRepair [22], JAI [4], SketchFix [18] \\
\hline
\textbf{No} & Other & LSRRepair [32], ssFix [61], DeepRepair [19], NPEFix [12] \\
\hline
\textbf{Yes} & Open-source & jGenProg [38], jKali [38], jMutRepair [38], Cardumen [39], DynaMoth [5], Nopol [64], ACS [63], SimFix [15], kPAR [29], FixMiner [19], AVATAR [30], TBar [31], ARJA [67], DynaMoth-A [67], Kali-A [67], RSRepair-A [67] \\
\hline
\end{tabular}
\end{table}

\*FixPR was not available before August 2019. \*\*SRepair relies on the data from the run-time GitHub repositories and needs a private deep learning model [28] and an online code search engine [15] to search syntactically- or semantically-similar code, which would be biased to assess its repair efficiency. \*sfix fails to execute as it relies on a private code search engine that is failed to connect. \*DeepRepair is not working, thus it is not selected. \*\*NPEFix is not selected as it does not use any fault localization technique.

ARJA is also a genetic programming approach to optimizing the exploration of the search space by combining three different approaches. RSRepair-A is a Java implementation of RSRepair [46], a Random-Search-based Repair tool, which tries to repair faulty programs with the same mutation operations as GenProg but uses random search, rather than genetic programming, to guide the patch generation process. SimFix utilizes code change operations from existing patches and similar code to build two search spaces, of which intersection is further used to search fix ingredients for repairing bugs. jKali and Kali-A are Java implementations of Kali [48] that fixes bugs with three actions: removal of statements, modification of if conditions to true/false, and insertion of return statements. jMutRepair implements the mutation-based repair approach [5] for Java programs, with three kinds of mutation operators (i.e., relational, logical and unary) to fix buggy if-condition statements.

\textit{Constraint-based repair approaches.} These approaches generally focus on fixing a single conditional expression that is more prone to defects than other types of program elements. Nopol [64], Dy-naMoth [8] ACS [63], and Cardumen [39] are dedicated to repairing buggy if conditions and to adding missing if conditions. Nopol relies on an SMT solver to solve the condition synthesis problem. DynaMoth leverages the runtime context, which is a collection of variable and method calls, to synthesize conditional expressions. ACS is proposed to refine the ranking of ingredients for condition synthesis. Cardumen repairs bugs by synthesizing patch candidates at the level of expressions with its mined templates from the program under repair to replace the buggy expression.

\textit{Template-based repair approaches.} These approaches are also often referred to as pattern-based and include kPAR [29], AVATAR [30], FixMiner [19] and TBar [31]. kPAR is the Java implementation of PAR [17] that repairs bugs with fix patterns manually summarized from human-written patches. FixMiner automatically mines fix patterns from the code repository for patch generation. AVATAR relies on the fix patterns for static analysis violations. TBar combines diverse fix patterns collected from the literature.

Note that, technically, template-based repair approaches can be viewed as heuristics-based approaches. In this study, however, we separate them in their category to highlight their specificity. Finally, there exist some repair approaches that are enhanced by machine learning techniques. Le Goues et al. [26] refer to them as learning-based repair approaches. One example of such approaches is the Prophet tool by Long and Rinard [35]: it learns from a corpus of code a model of correct code, which indicates how likely a given piece of code is w.r.t. the code corpus. Our criteria of subject selection

Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yes Le Traon
however excluded all learning-based repair as they are generally not “standalone”.

Our study considers the most diverse set of repair tools in the literature for a systematic assessment of APR. Notably, we cover different categories of repair approaches, while the previous record for a large scale study, which is held by Durieux et al. [7] on APR benchmark overfitting, did not consider the most widespread template-based tools. Furthermore, their study did not include ACS and SimFix from the current state-of-the-art in Java APR.

3.3 Experiment Settings

We now overview the inputs (i.e., buggy programs and fault localization information) and the validation process used in our study.

3.3.1 Defect Benchmark. The APR literature includes several benchmarks [16, 17, 36, 49]. In recent work, Durieux et al. showed that APR system may overfit the study benchmarks in terms of repairability. Since our objective is on efficiency, we focus on a single commonly used benchmark in the literature. We consider Defects4J [16] as it has been widely employed to assess approaches [15, 22, 32, 57], or to conduct various APR studies [34, 53, 55, 58], as well as other software engineering research [2, 3, 45, 47]. Defects4J consists of 395 bugs across six Java open source projects. Its dissection information [53] shows that the dataset contains a diversity of bug types. Our experiments thus consist of running each selected APR tool to generate patches in an attempt to fix each Defects4J bug. Overall, our experiments led to 347,603 repair attempts (each attempt requiring program compilation and testing against the test suite).

3.3.2 Fault Localization. As reported by Liu et al. [29], repair performance of APR tools could be biased by fault localization settings. To minimize such potential bias, we take on the challenge and implementation effort to re-configure all APR tools so that they are using the same fault localization information for each Defects4J bug. In our experiments, we employ the latest release of GZoltar v1.7.2, an on-hand test automation framework. Note that early versions of this tool were widely used in the APR community [15, 38, 57, 63]. However, Liu et al. revealed that the new version yields better results in the context of program repair [29]. For sorting suspicious statements, we use the Ochiai [1] ranking metric. Eventually, APR tools are fed with a ranked list of suspicious source code statements that should be changed within the buggy program to repair it.

3.3.3 Patch Validation. Patch validation is performed by APR systems based on the execution outcome of regression and bug-triggering test cases, i.e., test cases that are passed by the buggy program and those that, because they are not passed, reveal the existence of a bug. If a patch candidate can make the revised buggy program pass the entire test suite successfully, it is considered as a valid patch. Such a patch, however, could be incorrect if it is just overfitting the test suite [48, 62]. Thus, the community has adopted the terminology of plausible [48] patches to refer to patches that pass all test cases.

In recent literature, following the criticism on overfitting, researchers are shifting towards investigating correctness [20, 62]. So far, this has been a manual effort based on a recurrent criterion: a plausible patch is considered as correct when it is semantically similar to the developer’s patch in the benchmark. Unfortunately, the scope of semantics for APR is not explicitly defined as it is subjective.

### Table 3: Example rules that the community applies to confirm semantic similarity between tool-generated and developer-provided patches.

<table>
<thead>
<tr>
<th>Rule ID</th>
<th>Rule description</th>
<th>Illustrations</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Different hole with the same value (or alias)</td>
<td>if (fa * fb &gt; 0.0) { return foundDigit &amp;&amp; !hasExp; }</td>
</tr>
<tr>
<td>R2</td>
<td>Same exceptions but different messages</td>
<td>if (str == null) throw new NumberFormatException(str + “ is not a valid number.”);</td>
</tr>
<tr>
<td>R3</td>
<td>Variable initialization with new value rather than a default value</td>
<td>if (str == null) throw new StringIndexOutOfBoundsException();</td>
</tr>
<tr>
<td>R4</td>
<td>if statements instead of a ternary operator</td>
<td>if (!hasDecPoint==true) { return false; }</td>
</tr>
<tr>
<td>R5</td>
<td>Using a method</td>
<td>this.elitismRate = elitismRate;</td>
</tr>
<tr>
<td>R6</td>
<td>Replacing a value without a side effect</td>
<td>int g = (int) ((value - this.lowerBound) / (this.upperBound - this.lowerBound));</td>
</tr>
<tr>
<td>R7</td>
<td>Removing a statement</td>
<td>if (fa * fb &gt; 0.0) {</td>
</tr>
<tr>
<td>R8</td>
<td>Unnecessary code</td>
<td>w = -1; for(int i= 0; i&lt;value.length(); ++i) { if(Character.isWhitespace(c)) { } }</td>
</tr>
<tr>
<td>R9</td>
<td>Return outlier instead of a packaged return</td>
<td>return foundDigit &amp; return &amp; fixDePoint;</td>
</tr>
<tr>
<td>R10</td>
<td>More null checks</td>
<td>if ((hasDecPoint()==null) &amp;&amp; replacementInt()==null) { continue; }</td>
</tr>
</tbody>
</table>

We applied these rules to determine whether a plausible patch is a correct one when it is syntactically different from the patch that a developer wrote. In the second column, “tool name—bugID” denotes that the patch generated by the tool is identified as correct. The patches in the grey background are generated by APR tools while the patches in the white background are patches written by the developers.

We propose in this work to provide a first attempt of explicitly determining semantic similarity among patches. Our objective is to reduce the threat of subjectivity and enable reproducible experiments. To that end, we call on the community and consider labels of patches within APR research artifacts. We manually revisit patches that are generated by APR tools and which researchers have considered as correct in the literature. The objective is to unveil the implicit rules that researchers use to make the decisions on correctness. We find that there are broadly two scenarios when comparing a generated patch against the developer-provided patch:

1. **Identical patches**: in this case, the two patches are exactly identical, excluding variations in whitespace, layout, and comments.

2. **Semantically-similar patches**: in this case, the patches are not identical, although developers regard that they have the same effect on the program behavior. In Table 3 we summarize a taxonomy of correctness decision based on our study of patches labeled as correct by the research community. This taxonomy is based on the patches generated by ACS, SimFix, AVATAR, FixMiner, kPAR, and TBar whose authors investigated correctness and provided their manually labeled patches as research artifacts.

In the remainder of this paper, for the experiments with the 16 APR tools, we will systematically build on the rules of Table 3 to label plausible patches as correct. Thus, unless a generated patch is

---

*We enumerated only 10 rules in this paper due to space limitation. Please visit [https://github.com/SerVal-DTF/APR-Efficiency](https://github.com/SerVal-DTF/APR-Efficiency) for more rules and detailed descriptions.*
identical to the developer patch, it must fall under rules R1-10 to
be labeled as correct. Our rules are certainly not exhaustive neither
for defining semantic similarity nor for defining patch correctness.
We call on a community effort to augment these rules to enable
reproducible research. Due to space constraints, we only detail here a single rule. Con-
sider rule R5: In the illustration example, the developer patch en-
sures that boundaries are checked by calling a function that imple-
ments it. In contrast, a patch generated by ACS [63] directly inserts
the necessary code to check the boundary. Both patches, which are
not syntactically identical, are semantically similar. In the end, plausible and correct patches have the following relationship: Let P and C be sets of plausible and correct patches, respectively. It always holds C ⊆ P. We compute |C|/|P| as the Correct-
ness Ratio (CR) of generated plausible patches that are correct.

3.3.4 Halting Threshold. In the APR community, it is commonly accepted that patch generation processes are halted if a system runs out of the time budget before being able to find a valid patch. As discussed in Section 2, time can be a biased metric. Therefore, in this study, we propose to halt the repair systems by setting a threshold of repair attempts for a given bug. We set the threshold of attempts as 10,000. This number is selected based on the reported average number (9,696.5) of patch candidates generated by PraPR [12] for its fixed bugs. Given that PraPR works at the mutation level and does not require re-compilation, the number of attempts could be higher than that of other tools and it is high enough for the 16 APR tools employed in this study.

3.4 Terminology

Given that correct patches are first and foremost plausible patches, we propose in this work to use the term valid patches when referring to all plausible patches (including correct ones). Unless otherwise specified, we will also refer to as plausible all valid patches that have not yet been manually assessed as correct. We consciously avoid the term incorrect since the definition of correctness in Section 3.3.3 is sound, to some extent, but is not complete (i.e., there are some cases of semantic similarity that are missed).

3.5 Efficiency Metric: NPC

As motivated in Section 2, we employ as efficiency metric in this study the number of patch candidates (NPC) generated by APR tools until the first plausible patch is found. This metric was initially proposed by Qi et al. [47] as a proxy to measure the performance of fault localization techniques based on program repair tools. JAIID [4] and PraPR [12] recently used them to highlight the performance of their approaches. Nevertheless, efficiency has not been systematically assessed before. In this study, we further differentiate generated patches that turn out to be invalid into two groups:

(1) Nonsensical patch: Such a patch cannot even make the patched buggy program successfully compile [17, 40].
(2) In-plausible patch: Such a patch lets the patched buggy program successfully compile, but fails to pass some test cases in the available test suite.

Our efficiency metric is then computed by summing the number of patches in each category:

\[ NPC = NPC_{\text{nonsensical}} + NPC_{\text{in-plausible}} + NPC_{\text{valid}} \]

In practice, \( NPC_{\text{valid}} = 1 \) since the generation of patches is halted as soon as the first valid patch is found. In this study, since we aim to investigate the repair efficiency, we focus on bugs for which the repair attempts were successfully concluded. Thus, our experimental data do not mention the cases where many patch candidates are generated but none of them was valid. We leave this investigation as a future study.

4 STUDY RESULTS

We now provide experimental data as well as the key insights that are relevant to our research questions.

4.1 RQ1: Repairability Across Time

Table 4 provides execution outcomes of 16 repair tools on the Defect4J benchmark. We count the number of bugs that are plausibly fixed by each tool implementation, and further provide the number of plausible patches that can be considered as correct following the rules of patch validation (cf. Section 3.3.3).

Table 4: Numbers of Defect4J bugs that are correctly (plausibly) fixed by the different APR tools.

<table>
<thead>
<tr>
<th>APR Tool</th>
<th>C</th>
<th>Cl</th>
<th>L</th>
<th>M</th>
<th>Mc</th>
<th>T</th>
<th>Total</th>
<th>CR(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jGenProg</td>
<td>0</td>
<td>5</td>
<td>22</td>
<td>3</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>35</td>
</tr>
<tr>
<td>GenProxA</td>
<td>0</td>
<td>5</td>
<td>22</td>
<td>3</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>35</td>
</tr>
<tr>
<td>jParRepair</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>kPar</td>
<td>3</td>
<td>13</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RSRepairA</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>jKali</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Kali-A</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>DynaMoth</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Nopol</td>
<td>0</td>
<td>6</td>
<td>N/A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>ACS</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Cardumen</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>ARBA</td>
<td>1</td>
<td>10</td>
<td>2</td>
<td>29</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SimFix</td>
<td>3</td>
<td>18</td>
<td>5</td>
<td>1</td>
<td>10</td>
<td>5</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>FixMiner</td>
<td>5</td>
<td>14</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AVATAR</td>
<td>5</td>
<td>12</td>
<td>7</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>TBar</td>
<td>7</td>
<td>16</td>
<td>3</td>
<td>12</td>
<td>6</td>
<td>21</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

\( ^{\text{1}} \)The numbers outside the parentheses indicate the bugs fixed with correct patches while the numbers inside parentheses indicate the number of plausible patches. The missing numbers are marked with N/A as we failed to change the fault localization input for Closure program bugs for DynaMoth and Nopol, of which fault localization is tightly tied with GZGlib-0.0.1.C, Cl, L, M, Mc, and T represent Chart, Closure, Lang, Math, Mockito, and Time, respectively. The same as Table 5.

- [Template-based repair tools are the most effective.] We observe that kPar, FixMiner, AVATAR and TBar, which are template-based repair tools, present better repair performance than other tools in terms of the number of fixed bugs. The state-of-the-art, SimFix, also performs among the top. Note that, although it is classified as heuristics-based, and does not use templates explicitly, it performs transformations based on similar changes, and thus has been presented in previous studies [31] as template-based.

- [Patch ordering strategies are necessary to increase the likelihood of hitting correct patches.] Among the 16 repair tools, ACS exhibits the highest ratio of plausible patches that are found to be correct. This experimental finding confirms the strategy used by the authors to increase "precision" in patch generation: these are dependency-based ordering, document analysis, and predicate mining.
The intersection of tool X (row) and tool Y (column) contains the percentage of bugs fixed by tool X which are also fixed by tool Y. For instance, 40% of the bugs fixed by jGenProg (row 1) are also fixed by GenProg-A (column 2). On the contrary, 26.7% of the bugs fixed by GenProg-A (row 2) are also fixed by jGenProg (column 1). While the diagonal cells represent the number of bugs exclusively fixed by each repair tool. The intersection of tool X (row) and tool Y (column) contains the percentage of bugs fixed by tool X which are also fixed by tool Y. For instance, 40% of the bugs fixed by jGenProg (row 1) are also fixed by GenProg-A (column 2). On the contrary, 26.7% of the bugs fixed by GenProg-A (row 2) are also fixed by jGenProg (column 1). While the diagonal cells represent the number of bugs exclusively fixed by each repair tool. The intersection of tool X (row) and tool Y (column) contains the percentage of bugs fixed by tool X which are also fixed by tool Y. For instance, 40% of the bugs fixed by jGenProg (row 1) are also fixed by GenProg-A (column 2). On the contrary, 26.7% of the bugs fixed by GenProg-A (row 2) are also fixed by jGenProg (column 1). While the diagonal cells represent the number of bugs exclusively fixed by each repair tool.
produce the largest number of patch candidates. They are effective since they end-up finding a valid patch, but they are not efficient as they generate too many patches (comparing against other approaches) for repair attempts. On the other hand, constraint-based APR tools (e.g., ACS) have the lowest NPC scores. There is, therefore, an insight that constraint-solving and synthesis strategies, although they might require more computing effort to traverse the search space, eventually yield patches which waste less resource during test-based validation.

- [The state-of-the-art can avoid generating nonsensical patches.]

Figure 6 illustrates the contribution of nonsensical and in-plausible patches to the NPC scores. The distributions of nonsensical patches are interesting with respect to different claims in the literature. Indeed, to motivate their seminal work on template-based program repair, Kim et al. [17], authors of the PAR tool, stated that pioneer genetic programming based repair tools had the limitation that it could generate nonsensical patches. Our empirical assessment results back up this claim. However, our results also reveal that template-based repair tools (e.g., kPAR and TBar) have not fulfilled the claimed promise since they produce the largest numbers of nonsensical patches. This finding calls for a triaging strategy targeting the repair search space.

Among the template-based repair tools, kPAR has the smallest number of templates. Indeed it includes 10 templates manually pre-computed by Kim et al. [17], while AVATAR includes 11, TBar integrates 35 and FixMiner considers 28. Nevertheless, experimental results for NPC scores (cf. Figure 6) and the dissection in non-sensical and in-plausible categories (cf. Figure 7) reveal that kPAR is the least efficient. According to the authors’ source code of the tools, these tools use the same search space traversal strategy and implementation. Therefore, the only difference being about the included templates, we can safely conclude that the nature of these templates is driving the efficiency performance. AVATAR indeed focuses on templates obtained by curated datasets of fixes: all mined code changes are for static analysis violations which are systematically validated as actual fixes. FixMiner, on the other hand, augments its templates with relevant contextual information to ensure that they are applied on code locations that are syntactically similar to the locations where the templates were mined.

- [Specialized templates increase the efficiency of APR tools.]

Figure 7 illustrates the distribution of NPC scores for 16 APR tools. w.r.t. nonsensical patch generation is likely due to the combination of different search strategies that drive its genetic programming. SimFix heuristically searches similar code from the intersection of two search spaces: one is for donor code and the other one is for code change actions, to generate patches. A noteworthy result is that, while Nopol and DynaMoth overall generate few candidates, SimFix generates the largest number of patch candidates, none of which is ever found nonsensical. This finding suggests that code similarity has a large influence and can be useful for effectively triaging the repair space.

Besides Nopol, DynaMoth, and SimFix, five repair tools (i.e., jMutRepair, jKali, Kali-A, Cardumen and ARJA) generate significantly more in-plausible patches than nonsensical ones. jMutRepair, jKali and Kali-A are implemented with simple mutation operators that are unlikely to prevent the programs from compiling. However, these mutation operations can lead to test failures. ARJA's efficiency...
more abundant. We extend their study to consider the cases of in-plausible patches that are produced “before any plausible patch” (i.e., including if it is correct) vs. “before a correct patch” (i.e., only if the plausible is correct). Figure 8 illustrates the distributions of \( \text{NPC}_\text{in-plausible} \) scores for all fixed bugs and only correctly-fixed ones. We observe that for tools such as TBar, AVATAR, FixMiner, and kPAR, the median of \( \text{NPC}_\text{in-plausible} \) scores for correctly-fixed bugs is lower than the median for all fixed bugs. This means that, when a correct patch can be found, the number of in-plausible patches that are generated before is fewer than when only a plausible patch can be found. The situation is the converse for SimFix and ARJA. Therefore, we note that for most tools, a correct patch is more efficiently found when the search space is less noised (i.e., fewer in-plausible patches).

\[
\text{NPC} \quad \text{'All'} \quad \text{'Correct'} \quad \text{Median} \quad \text{Mean} \quad \text{# bugs}
\]

Table 6 provides more detailed statistics to drive an in-depth correlation study around efficiency and correctness. Based on the mean values, except for ACS, ARJA, and AVATAR, APR tools tend to generate more patch candidates when considering all bugs than when considering only the correctly-fixed ones. This tendency is much more apparent for \textit{search-based} APR techniques such as jGenProg [38], GenProg-A [67], SimFix [15], and RSRepair-A [67]. Although TBar is a template-based approach, it has characteristics of search-based tools since its search-space has been enlarged by incorporating any fix templates in the literature.

The previous experimental data overall suggest that simply giving more time to the APR tool to repair a buggy program does not guarantee to find correct patches. On the contrary, it seems that when allowing less attempts, the correctness ratio is improved. We propose to simulate a simple strategy of threshold setting to investigate the impact on the correctness ratio (i.e., ratio of correctly-fixed bugs to plausibly-fixed bugs). We consider a scenario where the APR tool is halted when a certain number of in-plausible patches is checked.

Figure 8: Number of in-plausible patch candidates generated before the first plausible patch.

Table 7 presents the results on how correctness ratio is influenced when we set a threshold on the number of in-plausible patches: basically, we propose to stop the repair attempts by a given tool if a certain number of generated patches turned out to be in-plausible (i.e., do not pass the test cases). We observe that the ratio of generated plausible patches to be correct is increased at varying degrees for 14 (out of 16) repair tools. Nopol and ACS do not show any improvement: initially, they produce few in-plausible patches. It should be noted that this result should be put in perspective as when discussing precision and recall: threshold setting, while useful to increase correctness ratio, may also lead to an overall reduction of the number of bugs that are correctly fixed.

\[
\text{NPC}_\text{in-plausible} \quad \text{threshold}.
\]

### 4.3 RQ3: Impact of Fault Localization Noise

A recent study by Liu et al. [29] has reported empirical results suggesting that fault localization results can adversely affect the performance of the repair. The authors experimented on a single tool, kPAR, and focused on reparability (i.e., how many bugs are not fixed due to localization errors). Our study already takes steps to avoid the bias of presenting various experimental results with APR tools which use different fault localization inputs. Thus, we
have put an effort to harmonize all fault localization configurations for the 16 APR tools under study (cf. Section 3.3.2).

To evaluate the impact of fault localization noise for different tools, we propose to compare results obtained so far with our standard spectrum-based fault localization (GZoltar+Ochiai) against experimental results where the APR systems are directly given the ground-truth fix locations. We compare the results both in terms of repairability and repair efficiency.

4.3.1 Impact of fault localization noise on repairability. First, we measure the impact on repairability, where we estimate for each repair tool how many bugs can be fixed by each APR system if it is precisely pointed to the ground-truth fix locations? Table 8 illustrates the details on the impact of repairability. Except for Cardumen, we observe that in general the correctness ratio improves (by up to 30 percentage points) if the fix locations are provided. It suggests that false-positive bug locations, hence fault localization noise, has an impact on the likelihood to generate correct patches. There are however anecdotal cases that are noteworthy:

- [Ground truth incompleteness.] Although our configuration of fault localization did not yield the developer-provided fix position for bug Lang-35, ACS patch generation eventually produced a correct patch for this bug. This patch, which targets a different code location, was found semantically-similar to the developer-provided rect patch for this bug. This patch, which targets a different code location, was found semantically-similar to the developer-provided fix position that is ranked by the fault localization. When the suspicious statement that is ranked by the fault localization is noisy (i.e., top suspicious statement(s) are false positives), more in-plausible and even non-sensical patches might be generated.

Table 8: Impact7 on repairability7 when ground-truth fix locations are directly given to the APR systems.

<table>
<thead>
<tr>
<th>APR Tool</th>
<th>C</th>
<th>L</th>
<th>M</th>
<th>Me</th>
<th>T</th>
<th>Total</th>
<th>CR (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jGitMiner</td>
<td>+30</td>
<td>+20</td>
<td>+10</td>
<td>+10</td>
<td>+10</td>
<td>+80</td>
<td>100</td>
</tr>
<tr>
<td>jMutRepair</td>
<td>+30</td>
<td>+20</td>
<td>+10</td>
<td>+10</td>
<td>+10</td>
<td>+80</td>
<td>100</td>
</tr>
<tr>
<td>jKali</td>
<td>+30</td>
<td>+20</td>
<td>+10</td>
<td>+10</td>
<td>+10</td>
<td>+80</td>
<td>100</td>
</tr>
<tr>
<td>jARJA</td>
<td>+30</td>
<td>+20</td>
<td>+10</td>
<td>+10</td>
<td>+10</td>
<td>+80</td>
<td>100</td>
</tr>
<tr>
<td>Cardumen</td>
<td>+30</td>
<td>+20</td>
<td>+10</td>
<td>+10</td>
<td>+10</td>
<td>+80</td>
<td>100</td>
</tr>
</tbody>
</table>

[7] This table shows variations of repairability w.r.t. results of our generic configuration of fault localization provided in Table 4. *(x+y)* means that if given exact fix locations, the tool can correctly fix *x* more bugs, but plausibly fixes *y* less bugs.

- [Fix location is different from bug location.] We observe that jKali now fails to correctly fix respectively 2 when it is given the developer-provided fix locations. This finding suggests that the repair tool is rather misled, in the cases of specific bugs, when it is given the right bug positions. Instead, some sibling positions are better inputs to drive correct fixing. However, data in Table 8 show fault localization has different impacts on performance for plausible fixing than for correct fixing.

Furthermore, based on results of overlapping in repairability (in terms of plausible patches) performance as depicted in Figure 9, we note that many bugs are only fixed (plausibly) when the fault localization does not precisely point to the fix locations. This is a surprising but interesting finding to be investigated by APR-targeted fault localization research.

- [Mockito bugs are not repairable.] Another immediate observation that we make from the experimental results in Table 8 is that bugs from the Mockito project are not easy to fix. According to reported results in Table 8, only three tools (i.e., FixMiner, AVATAR, and TBar) are able to fix Mockito bugs even if ground-truth fix locations are provided. We carefully proceed to investigate the possible reasons for this situation: 13 Mockito bugs (i.e., bug IDs 1-10 and 18-20) are associated to program code that cannot be compiled under JDK 7 (which is the JDK that is mentioned in the requirements of Defects4J). Our results further confirm a recent study [55] by Wang et al., who reported that the state-of-the-art SimFix and CapGen are not able to fix any Mockito bugs even when provided with ground-truth fix locations. Our study enlarges the scope of their studies. In the end, our systematic assessment results for all bugs better sheds light on a common phenomenon in the literature where Mockito project bugs are not considered when reporting repair performance. These results call for modular configuration of execution environment as well as for better integration of advances in fault localization to support APR systems. Besides Mockito bugs, many bugs in other projects cannot be fixed since they are not precisely localized. Overall, consider again Figure 9. For all tools (except jMutRepair), we observe that some bugs are fixed only when the actual fix locations are directly given to the system.

4.3.2 Impact of fault localization noise on repair efficiency. We investigate the NPC scores, i.e., the number of patch candidates that are generated by the different APR systems when they are pointed to the developer-provided fix locations. Figure 10 shows the corresponding distribution of NPC scores for each repair tool.

- [Template-based program repair tools are highly sensitive to fault localization noise.] We observe from Figure 10 that, except for DynaMoth, Nopol, and ACS, the remaining 13 repair tools have significantly smaller distribution ranges of NPC scores than the distribution ranges when the APR system was run under our generic fault localization configuration (cf. Figure 6). A straightforward explanation is that, under a typical fault localization configuration, a repair tool will attempt to generate patch candidates for each suspicious statement that is ranked by the fault localization. When the fault localization is noisy (i.e., top suspicious statement(s) are false positives), more in-plausible and even non-sensical patches might be generated. In particular, for repair tools that are based on pattern matching and code similarity (i.e., SimFix, and the template-based repair tools) the gap of repair efficiency has reduced substantially by an order of magnitude when correct fix locations are given to the tool. For example, the median NPC score of SimFix is around 200 when using our generic configuration of fault localization, but is around 20 when using directly correct fix locations. Such tools are thus more sensitive to fault localization noise than other tools. In conclusion, we confirm the finding of the study of Liu et al. [29]. However, we delimitate its validity to template-based repair tools. Other tools, e.g., constraint-based repair tools such as ACS or
Nopol, which use specific techniques to triage the search space do not present any increase in repair efficiency when pointed to the fix locations. This finding suggests that they have limited sensitivity to fault localization noise.

6 RELATED WORK

Performance Evaluation. Initially, evaluation of test-based program repair was focused on counting the number of bugs fixed by a repair tool out of all bugs in a benchmark [17, 22, 25, 56]. However, valid patches are sometimes incorrect as they overfit on incomplete test suites [48], and might cause issues during maintenance [10, 52]. Thus, plausibility and correctness became widely accepted to define metrics for assessing repairability of repair tools [4, 12, 14, 19, 29–32, 50, 51, 57, 63]. In this study, we also follow the metric to revisit the repairability of repair tools. Nevertheless, we differ from studies in the literature by ensuring that APR tools use the same controlled configuration for fault localization.

Repair Efficiency. Along with the performance evaluation, several studies simply reported the repair efficiency in terms of CPU time consumption of fixing bugs [12, 14, 56, 57, 63]. However, it could be biased to assess the efficiency with time cost for various reasons (cf. Section 2). Instead, we leverage the number of patch candidates generated by repair tools to measure the repair efficiency, which should be intrinsic to the repair approaches. Ghanbari et al. [12] provided information on the number of patch candidates generated by PraPR. This information, however, could not be put into perspective against other tools. Our study fills this gap.

5 THREATS TO VALIDITY

External validity. Our study considers only the Defects4J benchmark and only java repair tools. All findings might thus be valid only for this configuration. Nevertheless, this threat is mitigated by the fact that we use a large set of repair tools and a renowned defect benchmark to study a performance criterion that was largely ignored in the literature.

Internal validity. Our implementation of fault localization as well as the manual assessment of patch correctness may threaten the validity of some of our conclusions. We mitigate this threat by reusing common fault localization components from the repair literature as well as by enumerating and sharing the rules for defining patch correctness. Two authors were in charge of assessing the correctness and they cross-reviewed each other’s decisions. In case of conflict other authors were called to create a consensus.

Construct validity. By construct, to limit resource exhaustion, we added a threshold on the number of patches to validate. However, this threshold may penalize some tools. We mitigate this threat by carefully selecting a threshold based on empirical results on PraPR, a recent related work which mutates directly bytecode, allowing it to generate many more patches (since no compilation is needed).

7 CONCLUSION

This paper reports on a large-scale study on the efficiency of test suite based program repair. Efficiency is defined based on the number of patch candidates that are generated before a repair system can hit a valid patch. Our study comprehensively runs 16 repair systems from the literature under identical configuration of fault localization. Our experiments explore repairability (i.e., repair effectiveness), repair efficiency as well as the impact of fault localization on both performance criteria. Beyond the statistical data, we call on the community to invest in strategies for making repair efficient in order to facilitate adoption in a software industry where computing resources are managed sometimes with parsimony.

Artefacts: All data and tool support for replication are available at https://github.com/SerVal-DTF/APR-Efficiency.git

ACKNOWLEDGMENTS

This work is supported by the Fonds National de la Recherche (FNR), Luxembourg, through RECOMMEND 15/IS/10449467 and FIXPATTERN C15/IS/9964569, and is supported through the National Natural Science Foundation of China No. 61672529.
REFERENCES


[38] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault
On the Efficiency of Test Suite based Program Repair

- Shangwen Wang, Ming Wen, Xiaoguang Mao, and Deheng Yang. 2019. Attention
- Shangwen Wang, Ming Wen, Liqian Chen, Xin Yi, and Xiaoguang Mao. 2019. How
- Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and
- Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The
- Edward K Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure
- Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. 2013. Using automated
- Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. 2014. The
- Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. 2013. Using automated

References:


