On the Efficiency of Test Suite based Program Repair

A Systematic Assessment of 16 Automated Repair Systems for Java Programs

Anonymous Author(s)

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. While 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. We focus on assessing 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 *sensical* 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 run under the same fault localization configurations to limit biases. Eventually, among other findings, we note that: (1) many irrelevant patch candidates are generated by repair tools 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.

KEYWORDS

Patch generation, Program repair, Efficiency, Empirical assessment.

1 INTRODUCTION

In the last decade, Automated Program Repair (APR) [11, 26, 42] has extensively grown as a prominent research topic in the software engineering community. Figure 1 overviews the research activity in 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 [40], which shows that the research in this field has been revived with the seminal work, ten years ago, of Weimer et al. [55] 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].

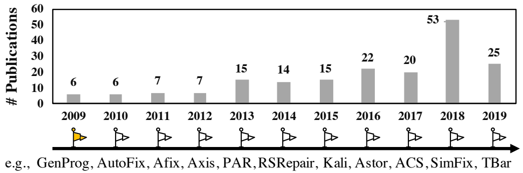

Figure 1: APR research publications since 2009[1].

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 suites) 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 assessments reports [12, 62]. Yet, efficiency is a key property for bringing program repair into general use within practitioners' settings. Indeed, APR aims to alleviate the manual effort involved in resolving software bugs, and holds this promise in two scenarios: in production, it is expected to drastically reduce the time-to-fix delays and minimize downtime; in a development cycle, APR can help suggest changes to accelerate debugging. Yet, until now, literature approaches [15, 31, 31, 51, 62] have mainly focused on highlighting the increased performance on *eventually* fixing more and more benchmark bugs. In recent work, Ghanbari et al. [12] raised the efficiency issue and built on the time cost criterion to demonstrate the efficiency of their PraPR tool (which does not require re-compiling source code). This criterion, which was already mentioned in a

ICSE, 23 - 29 May 2020, Seoul, South Korea
2020. ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

[1] Data are extracted from Monperrus's living review on APR [40].

few of the previous work [33, 56, 62], 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 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 fill a gap in the literature of comprehensive analysis of repair efficiency in terms of generated patches for a large spectrum of APR systems. Eventually, we gather insights on how literature approaches' strategies impact repair efficiency. Overall, we mainly find that:

- F0: So far, efficiency is not a widely-valued performance target. We found that the 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 (approximatively) 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 regarded as a **valid** patch. This criterion was first used by Weimer et al. [55] in their seminal work on GenProg, and has become the de-facto metric of repair performance [26]. Nevertheless, as later studies have revealed, even if a generated patch can pass all test cases, it might break a necessary behaviour or introduce other faults, which are not covered by the given test suite [52]. Besides, a developer may not accept the patch due to several reasons such as coding convention [17, 41]. All such valid patches in terms of the test suite are therefore now referred to as **plausible** since they require further investigations to ensure that they are **correct**, i.e., acceptable to developers. In the literature, *correctness* is generally assessed manually by comparing the APR-generated patch against the developer-provided patch available in the benchmark.

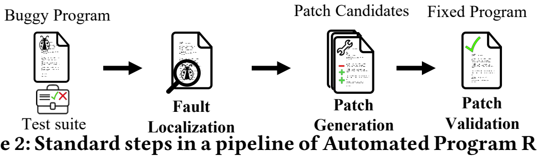

Figure 2: Standard steps in a pipeline of Automated Program Repair.

Studies in the literature, such as the recent work of Durieux et al. [7] on benchmark overfitting, generally focus on information about plausible patches given that correctness is hard to assess. Our work is the first to explore artifacts from the literature, where researchers provide correctness labels of their generated patches, in order to extract and categorize implicit rules used by the community to define correctness. We expect that these rules will be studied and augmented by the community to enable systematic assessment of correctness.

Efficiency of APR tools has been assessed in the literature [12, 14, 56, 62] via measuring the time-to-generate-and-validate patches. Table 1 presents the time cost of the PraPR [12] state-of-the-art repair tool on Defects4J [16] program samples. On average, for each Closure bug, PraPR generated and validated over 29 thousand patches, about 10 times more than the average number of patches that are generated and validated for each Chart bug. Yet, the time cost for Closure bugs is 20 times more than the time cost for Chart bugs. This suggests that it is challenging to define a generically-suitable time budget for repairing bugs. We further note that correlation tests did not reveal any linear correlation between the time cost of repairing a bug and benchmark properties such as the number of test cases or program sizes. Consequently, time cost may not be a reliable metric for efficiency.

Table 1: Average PraPR time cost (s) & # patches per bug [12].

Subjects	# Validated Patches	Time cost (s)
Chart	2,827.6	157.8
Closure	29,849.9	3,027.3

To further highlight the biases that execution times may carry, we refer to literature settings of time budgets for running APR systems: ACS [62] and SimFix [15] are evaluated with repair time budgets of 30 minutes and 5 hours, respectively. Furthermore, in [15], assessment comparison between ACS and SimFix does not consider the bias related to the difference in the execution platforms. A comparison of performance (in terms of how many bugs each tool can fix) may, therefore, be misleading: a given bug may have been fixed by one tool because the time budget is sufficient while it cannot be fixed by the other due to lack of time.

With two simple experimental runs of compiling and testing Defects4J samples, we confirm our concerns: time budgets could introduce biases for different bugs. Indeed, as revealed in Figure 3, different machine configurations may lead to drastically divergent compiling and testing time: irrespectively of projects, the MannâĂŞWhitneyâĂŞWilcoxon tests [37, 59] confirm that the first machine consumes statistically significantly more CPU time than the second machine either for compilation or for testing Defects4J buggy programs. These results definitively suggest that time cost is not a reliable metric to enable reproducible and comparable experiments on the efficiency of program repair.

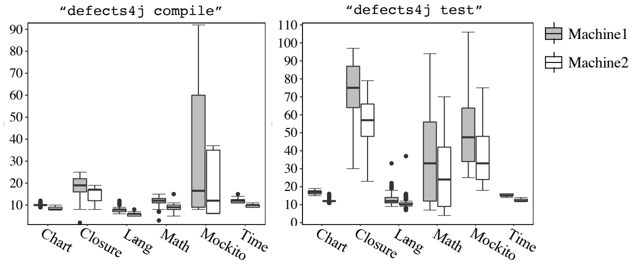

Figure 3: Distribution CPU times for compiling and testing Defects4J programs.

- ● Machine 1 runs OS X El Capitan 10.11.6 with 2.5 GHz Intel Core i7, 16GB 1600MHz DDR3 RAM.
- ● Machine 2 runs macOS Mojave 10.14.1 with 2.9 GHz Intel Core i9, 32 GB 2400MHz DDR4 RAM.

Instead, we propose to rely on the metric of **number of generated patch candidates**, which should be intrinsic to the approach and agnostic of machine configuration variabilities.

3 STUDY DESIGN

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/offline 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 undermine efficiency: nonsensical and inplausible patches (see Section 3.5).

(3) **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 for two repair systems the bias of fault localization on repairability.

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) *Availability*: our study involves the execution of APR tools, thus APR approaches without publicly available tools are excluded.

(2) *Executability*: some APR approaches provide publicly available tools, which however cannot be executed as-is for diverse issues (e.g., ssFix [60] failed to execute because an online connection to a private search engine fails). We exclude such approaches from the study.

(3) *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) *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 which would require extra data is excluded (e.g., LSRepair [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 [40]. 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 retained. 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.

Heuristic-based repair approaches. These approaches construct and iterate over a search space of syntactic program modifications [25]. Associated tools include **jGenProg** [38], **GenProg-A** [66], **ARJA** [66], **RSRepair-A** [66], **SimFix** [15], **jKali** [38], **Kali-A** [66], and **jMutRepair** [38]. jGenProg and GenProg-A are Java implementations of GenProg [55], which generates patches by searching donor code from existing code with the genetic programming method. 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,

Table 2: Included and excluded APR tools for our study.

Selected	Reason	APR Tools for Java Programs
No	Not public	PAR [17], xPAR [22], JFix/S3 [21], ELIXIR [50], Hercules [51], SOFix [33], CapGen [56], PraPR [12].
No	Faulty method required	HDRepair [22], JAID [4], SketchFix [14].
No	Other	LSRepair[*] [32], ssFix[★] [60], DeepRepair[†] [58], NPEFix[‡] [6].
Yes	Open-source & working	jGenProg [38], jKali [38], jMutRepair [38], Cardumen [39], DynaMoth [8], Nopol [63], ACS [62], SimFix [15], kPAR [29], FixMiner [19], AVATAR [30], TBar [31], ARJA [66], GenProg-A [66], Kali-A [66], RSRepair-A [66].

[*] LSRepair relies on the data from the run-time GitHub repositories and needs a private deep learning model [28] and an online code search engine [18] to search syntactically- or semantically-similar code, which would be biased to assess its repair efficiency. [★] ssFix fails to execution 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.

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.

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** [63], **DynaMoth** [8] **ACS** [62], and **Cardumen** [39] are dedicated to repairing buggy `if` conditions and to add missing `if` preconditions. 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.

Template-based repair approaches. These approaches are also often referred to as pattern-based and include **kPAR** [29], **FixMiner** [19], **AVATAR** [30] and **TBar** [31]. kPAR is the Java implementation of PAR that repair 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 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 focus 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, 31, 32, 56], or to conduct various APR studies [34, 53, 54, 57], 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, 56, 62]. 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, 61]. 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, 61]. 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.

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:

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

Rule ID	Rule description	Illustrations

R1	Different fields with the same value (or alias)	`-    return cAvailableLocaleSet.contains(locale);`
		`+    return availableLocaleList().contains(locale);`

	e.g., AVATAR→Chart_7	`+    return cAvailableLocaleList.contains(locale);`

R2	Same exception but different messages	`+    throw new NumberFormatException(str +`
		`                 " is not a valid number.");`

	e.g., ACS→Time_15	`+    throw new NumberFormatException();`
R3	Variable initialization with new rather than a default value	`+    if (str == null) str = "";`
	e.g., TBar→Lang_47	`+    if (str == null) str = new String();`
R4	`if` statement instead of a ternary operator	`+    classes[i] = array[i] == null ? null : array[i].getClass();`

	e.g., TBar→Lang_33	`+    if (array[i] == null) continue;`
		`+    classes[i] = array[i].getClass();`

R5	Unrolling a method	`-    this.elitismRate = elitismRate;`
		`+    setElitismRate(elitismRate);`

	e.g., ACS→Math_35	`+    if (elitismRate>(double)1.0){throw ...;}`
		`+    if (elitismRate<(double)0.0){throw ...;}`

R6	Replacing a value without a side effect	`-    int g = (int) ((value - this.lowerBound) / (this.upperBound`
		`+    int g = (int) ((v - this.lowerBound) / (this.upperBound`

	e.g., FixMiner→Chart_24	`-    v = Math.min(v, this.upperBound);`
		`+    value = Math.min(v, this.upperBound);`

R7	Enumerating	`-    if (fa * fb >= 0.0 ) {`
		`+    if (fa * fb > 0.0 ) {`

	e.g., ACS→Math_85	`+    if (fa * fb >= 0.0 &&!(fa * fb==0.0))`

R8	Unnecessary code uncleaned	`- boolean wasWhite= false;`
		`  for(int i= 0; i<value.length(); ++i) {`
		`-    if(Character.isWhitespace(c)) { ...... }`
		`-    wasWhite= false;`

	e.g., AVATAR→Lang_10	`-    if(Character.isWhitespace(c)) { ...... }`
		`-    wasWhite= false;`

R9	Return earlier instead of a packaged return	`-    return foundDigit && !hasExp;`
		`+    return foundDigit && !hasExp && !hasDecPoint;`

	e.g., ACS→Lang_24	`+    if (hasDecPoint==true){return false;}`

R10	More null checks	`+    if (searchList[i] == null || replacementList[i] == null)`
		`+    {    continue;    }`

	e.g., SimFix→Lang_39	`+    if(noMoreMatchesForReplIndex[i]||searchList[i]==null`
		`+    ||searchList[i].length()==0||replacementList[i]==null)`
		`+    {    continue;    }`

We applied these rules to determine whether a plausible patch is a correct one if it is syntactically different from the patch that a developer wrote. In the 2nd 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.

(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[2] to label plausible patches as correct. Thus, unless a generated patch is 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. Consider rule R5: In the illustration example, the developer patch ensures that boundaries are checked by calling a function that implements it. In contrast, a patch generated by ACS [62] 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 $C \subseteq P$. We compute $\frac{|C|}{|P|}$ as the ***Correctness 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[3], 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. JAID [4] and PraPR [12] recently used them to highlight the performance of their approach. 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, 41].

(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:

$$NPC = NPC_{nonsensical} + NPC_{in-plausible} + NPC_{valid}$$

In practice, $NPC_{valid} == 1$ 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 does not mention the cases where many patch candidates are generated but none of them was valid. We leave this investigation as a future study.

[2] We enumerated only 10 rules in this paper due to space limitation. Please visit https://github.com/APRStudy/APRStudy for more rules and detailed descriptions.

[3] The developer-patch provided in the benchmark, which we use as ground truth, may be erroneous as well.

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 Defects4J 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 Defects4J bugs that are correctly (plausibly) fixed by the different APR tools.

APR Tool	Chart	Closure	Lang	Math	Mockito	Time	Total	CR(%)
jGenProg	0 (5)	0 (2)	0 (2)	3 (11)	0 (0)	0 (0)	3 (20)	15
GenProg-A	0 (5)	2 (15)	0 (1)	0 (9)	0 (0)	0 (0)	2 (30)	6.7
jMutRepair	1 (4)	2 (5)	0 (2)	2 (11)	0 (0)	0 (0)	5 (22)	22.7
kPAR	4 (13)	2 (10)	0 (18)	4 (22)	0 (0)	0 (1)	10 (63)	15.9
RSRepair-A	0 (4)	2 (22)	0 (3)	0 (12)	0 (0)	0 (0)	2 (41)	4.9
jKali	0 (4)	1 (8)	1 (4)	1 (9)	0 (0)	0 (0)	3 (25)	12
Kali-A	0 (6)	2 (48)	0 (0)	1 (10)	0 (1)	0 (0)	3 (**65**)	4.6
DynaMoth	0 (6)	N/A	0 (2)	1 (13)	0 (0)	0 (1)	1 (22)	4.5
Nopol	1 (6)	N/A	1 (6)	0 (18)	0 (0)	0 (1)	2 (31)	6.5
ACS	2 (2)	0 (0)	3 (3)	11 (16)	0 (0)	1 (1)	17 (22)	**77.3**
Cardumen	1 (4)	0 (2)	0 (0)	1 (6)	0 (0)	0 (0)	2 (12)	16.7
AJRA	1 (10)	2 (29)	1 (3)	4 (15)	0 (1)	0 (0)	8 (58)	13.8
SimFix	2 (8)	7 (19)	5 (16)	10 (25)	0 (0)	0 (0)	**24 (68)**	**35.3**
FixMiner	5 (14)	0 (2)	1 (2)	7 (15)	0 (0)	0 (0)	13 (33)	**39.4**
AVATAR	5 (12)	7 (15)	4 (13)	3 (17)	0 (0)	0 (0)	**19** (57)	33.3
TBar	7 (16)	3 (12)	6 (21)	8 (23)	0 (0)	0 (0)	**24 (72)**	30.8

*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 GZoltar-0.0.1. The same as Tables 7 and 8.

• [*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"[4] in patch generation: these are dependency-based ordering, document analysis, and predicate mining.

• [*Through time, repair tools tend to subsume their predecessors in terms of which bugs are fixed*] Table 5 provides statistics on the percentage of fixed bugs that are overlapping between two repair tools. In this table, the tools in column headers and row headers are ordered chronologically concerning the date of approach publication. Note that jGenProg ranked based on the GenProg publication year although the tool itself was implemented years later. We note that the upper-right side of the table is relatively darker than the rest: the percentages of overlapping are higher for these cells. These results suggest that, overall, the bugs that are fixed by earlier tools are also generally covered by more recent tools. Besides, evolution trends presented in Figure 4 show that, although the number of bugs that are fixed by the different tools over the years is increasing, the number of new bugs is increasing with small increments. This result suggests that the strategies implemented in new approaches tend to have similar outcomes as merging past techniques to cover previous bug sets that were fixed each via different approaches.

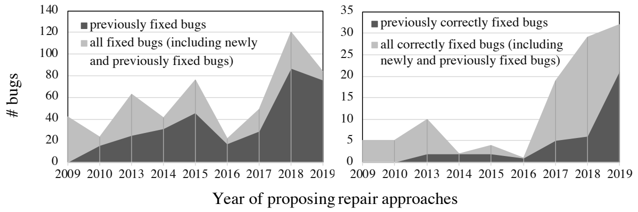

Figure 4: Evolution of the number of fixed bugs across time.

• [*Recent APR tools tend to correctly fix more bugs than their predecessors*] In the right part of Figure 4, a visible breakthrough is the sharp increase of the light grey area indicating that recent tools increasingly correctly fix bugs which have not been fixed by previous tools. We further summarize in Figure 5 the number of bugs that each tool can correctly fix exclusively or not. SimFix, TBar, and ACS are leading repair tools that generate correct fixes for more bugs. In contrast, jGenProg, GenProg-A, jMutRepair, RSRepair-A, jKali, Kali-A, DynaMoth, and Cardumen do not correctly fix any Defects4J bug that is not also correctly fixed by another tool.

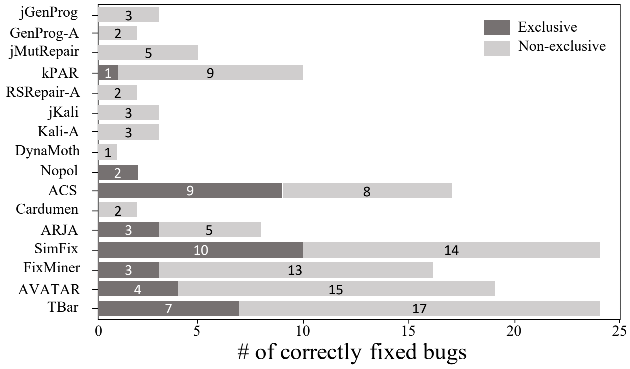

Figure 5: Repairing exclusivity of each APR tool (correct patches).

• [*Implementation details can make a difference*] Finally, we observe that Java-targeted implementations of GenProg (i.e, jGenProg and GenProg-A) and Kali (i.e., jKali and Kali-A) by different research groups yield diverging repair performance on the same benchmark.

Overall the systematic study of repairability of APR tools across time reveals that (1) recent tools tend to fix more bugs than their predecessors; (2) each newly proposed repair tool however plausibly fix few bugs that were not fixed by other tools; (3) more bugs can be correctly-fixed by lately APR tools; (4) template-based repair tools are the most effective to eventually produce plausible patches.
It thus remains unclear whether the strategies proposed by record-setting tools are improving the state-of-the-art of patch generation. We propose to focus on efficiency as a complementary metric to assess performance gains.

4.2 RQ2: Patch generation efficiency

Following our motivation argument in Section 2, we use the *NPC* scores (i.e., number of generated patch candidates that are checked until a valid patch is found) to measure repair efficiency of APR tools. For each tool, the results focus on Defects4J bugs that are fixed (i.e., a valid patch was eventually found). Indeed, through efficiency, we attempt to **measure the ability of the APR tool to**

[4] Precision is the terminology employed by its authors to refer to the ratio of correct patches to plausible patches.

Table 5: Number of overlapped fixed bugs per repair tool.

	jGenProg	GenProg-A	jMutRepair	kPAR	RSRepair-A	jKali	Kali-A	DynaMoth	Nopol	ACS	Cardumen	ARJA	SimFix	FixMiner	AVATAR	TBar
jGenProg	5.0% (1)	40.0% (8)	45.0% (9)	55.0% (11)	45.0% (9)	40.0% (8)	40.0% (8)	35.0% (7)	25.0% (5)	20.0% (4)	30.0% (6)	60.0% (12)	80.0% (16)	45.0% (9)	60.0% (12)	85.0% (17)
GenProg-A	26.7% (8)	0.0% (0)	36.7% (11)	46.7% (14)	90.0% (27)	33.3% (10)	80.0% (24)	23.3% (7)	20.0% (6)	16.7% (5)	10.0% (3)	96.7% (29)	40.0% (12)	30.0% (9)	43.3% (13)	53.3% (16)
jMutRepair	40.9% (9)	50.0% (11)	4.5% (1)	68.2% (15)	50.0% (11)	59.1% (13)	54.4% (12)	31.8% (7)	22.7% (5)	18.2% (4)	13.6% (3)	63.6% (14)	77.3% (17)	45.5% (10)	86.4% (19)	90.9% (20)
kPAR	17.5% (11)	22.2% (14)	23.8% (15)	6.3% (4)	25.4% (16)	25.4% (16)	25.4% (16)	22.2% (14)	25.4% (16)	11.1% (7)	7.9% (5)	39.7% (25)	49.2% (31)	34.9% (22)	57.1% (36)	74.6% (47)
RSRepair-A	22.0% (9)	65.9% (27)	26.8% (11)	39.0% (16)	2.4% (1)	26.8% (11)	75.6% (31)	19.5% (8)	22.0% (9)	12.2% (5)	7.3% (3)	85.4% (35)	29.3% (12)	19.5% (8)	39.0% (16)	41.5% (17)
jKali	32.0% (8)	40.0% (10)	52.0% (13)	64.0% (16)	44.0% (11)	8.0% (2)	56.0% (14)	40.0% (10)	24.0% (6)	8.0% (2)	12.0% (3)	56.0% (14)	56.0% (14)	20.0% (5)	76.0% (19)	68.0% (17)
Kali-A	12.3% (8)	36.9% (24)	18.5% (12)	24.6% (16)	47.7% (31)	21.5% (14)	23.1% (15)	13.8% (9)	9.2% (6)	3.1% (2)	1.5% (1)	63.1% (41)	21.5% (14)	15.4% (10)	29.2% (19)	27.7% (18)
DynaMoth	31.8% (7)	31.8% (7)	31.8% (7)	63.6% (14)	36.4% (8)	45.5% (10)	40.9% (9)	0.0% (0)	54.5% (12)	13.6% (3)	9.1% (2)	50.0% (11)	54.5% (12)	50.0% (11)	54.5% (12)	59.1% (13)
Nopol	16.1% (5)	19.4% (6)	16.1% (5)	51.6% (16)	29.0% (9)	19.4% (6)	19.4% (6)	38.7% (12)	19.4% (6)	12.9% (4)	6.5% (2)	25.8% (8)	25.8% (8)	19.4% (6)	38.7% (12)	35.5% (11)
ACS	18.2% (4)	22.7% (5)	18.2% (4)	31.8% (7)	22.7% (5)	9.1% (2)	9.1% (2)	13.6% (3)	18.2% (4)	40.9% (9)	13.6% (3)	36.4% (8)	22.7% (5)	18.2% (4)	31.8% (7)	40.9% (9)
Cardumen	50.0% (6)	25.0% (3)	25.0% (3)	41.7% (5)	25.0% (3)	25.0% (3)	8.3% (1)	16.7% (2)	16.7% (2)	25.0% (3)	8.3% (1)	25.0% (3)	58.3% (7)	50.0% (6)	50.0% (6)	83.3% (10)
ARJA	20.7% (12)	50.0% (29)	24.1% (14)	43.1% (25)	60.3% (35)	24.1% (14)	70.7% (41)	19.0% (11)	13.8% (8)	13.8% (8)	5.2% (3)	6.9% (4)	31.0% (18)	25.9% (15)	39.7% (23)	43.1% (25)
SimFix	23.5% (16)	17.6% (12)	25.0% (17)	45.6% (31)	17.6% (12)	20.6% (14)	20.6% (14)	17.6% (12)	11.8% (8)	7.4% (5)	10.3% (7)	26.5% (18)	19.1% (13)	25.0% (17)	39.7% (27)	58.8% (40)
FixMiner	27.3% (9)	27.3% (9)	30.3% (10)	66.7% (22)	24.2% (8)	15.2% (5)	30.3% (10)	33.3% (11)	18.2% (6)	12.1% (4)	18.2% (6)	45.5% (15)	51.5% (17)	9.1% (3)	54.5% (18)	75.8% (25)
AVATAR	21.1% (12)	22.8% (13)	33.3% (19)	63.2% (36)	28.1% (16)	33.3% (19)	33.3% (19)	21.1% (12)	21.1% (12)	12.3% (7)	10.5% (6)	40.4% (23)	47.4% (27)	31.6% (18)	5.3% (3)	78.9% (45)
TBar	23.6% (17)	22.2% (16)	27.8% (20)	65.3% (47)	23.6% (17)	23.6% (17)	25.0% (18)	18.1% (13)	15.3% (11)	12.5% (9)	13.9% (10)	34.7% (25)	55.6% (40)	34.7% (25)	62.5% (45)	5.6% (4)

The intersection of tool X (row) and tool Y (column) contains the percentage of bugs fixed by X which are also fixed by 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 present the number of bugs exclusively fixed by each repair tool.

avoid wasting computing resource, time and energy in patch validation towards generating a valid patch.

Figure 6 overviews the general distributions of *NPC* scores of the 16 repair tools on the Defects4J benchmark. For all tools, the median *NPC* is lower than 250 patch candidates. However, the distribution spread among bugs is not only significant for several (8 out of 16) tools but also vary across tools.

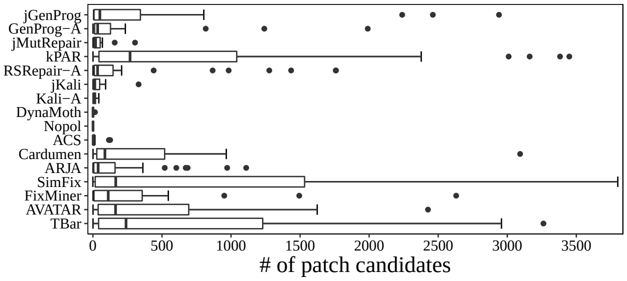

Figure 6: The distribution of NPC scores for 16 APR tools.

•[*Efficiency is not yet a widely-valued performance target.*] SimFix, TBar and kPAR exhibit the highest *NPC* scores which can go beyond 1,000 patch candidates for some bugs. Correlating this data with repairability findings (Section 4.1), we note that tools with highest repairability scores also have the highest *NPC* scores (hence, *lower efficiency*). In particular, we note that APR approaches which rely on change patterns (i.e., standard template-based tools) or heuristically search for donor code based on code similarity (e.g., SimFix) produces 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 7 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 nonsensical patches within the search space. In this regard, our experimental results highlight three tools (i.e., DynaMoth, Nopol, and SimFix) which do not generate any nonsensical patches.

Nopol uses an SMT solver to address the condition patch synthesis problem. DynaMoth leverages the runtime context, collects variable and method calls to synthesize conditional expression patches. 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 search 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 w.r.t. nonsensical patch generation is likely due to the combination of different search strategies that drive its genetic programming.

• [*The more templates an APR system considers, the more nonsensical and in-plausible patches it will generate.*] TBar contains more fix templates than kPAR, FixMiner and AVATAR, since it merges all literature templates. Therefore, each suspicious buggy location has a higher probability in TBar to be matched with more templates, leading to more patch candidates than other tools. This finding highlights the importance of strategies for fix template matching and donor code searching to improve the repair efficiency of template-based repair tools.

• [*Specialized templates increase the efficiency of APR tools.*] Among the template-based repair tools, kPAR has the smallest number of templates. Indeed it includes 10 templates manually prepared 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,

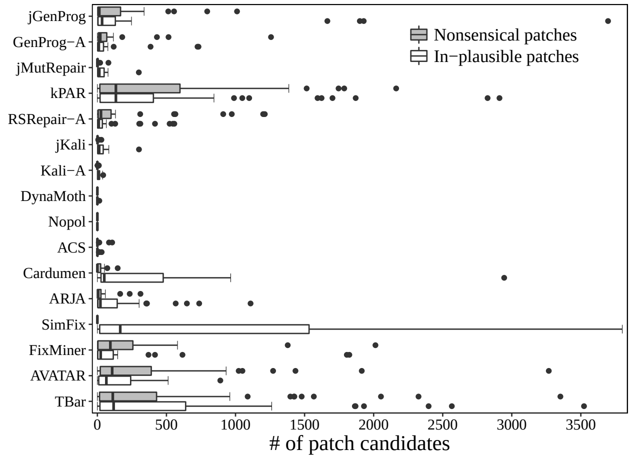

Figure 7: Distributions of $NPC_{nonsensical}$ and $NPC_{in-plausible}$ scores for each APR tools.

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 where mined.

• [*Correct patches are sparse in the search space.*] Long et al. [34] presented an initial study which revealed that correct patches can be considered as sparse in the search space and that overfitting patches [20, 23, 48, 61] (i.e., only plausible but not correct) are vastly 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 $NPC_{in-plausible}$ 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 $NPC_{in-plausible}$ 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).

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 *search-based* APR techniques such as jGenProg [38], GenProg-A [66], SimFix [15], and RSRepair-A [66]. Although TBar is a template-based approach, it has characteristics of search-based tools since its search-space have been enlarged by incorporating **any** fix templates in the literature.

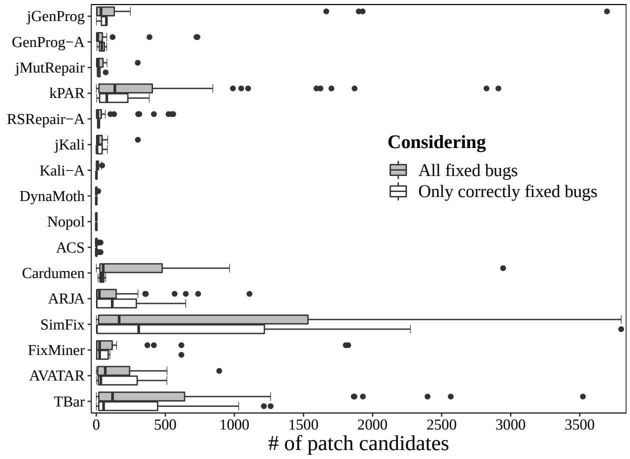

Figure 8: Number of In-plausible patch candidates generated before the first plausible patch.

Table 6: Upper whisker, median and mean values of NPC ($NPC_{in-plausible}$) scores in Figure 8.

APR Tools	Upper Whisker		Median		Mean		#
	All	Correct	All	Correct	All	Correct	bugs
jGenProg	803 (247)	191 (79)	50 (34)	↑ 127 (73)	670 (436)	108 (51)	3
GenProg-A	235 (76)	139 (40)	34 (11)	↑ 75 (41)	187 (81)	75 (40)	2
jMutRepair	67 (77)	33 (27)	20 (14)	↑ 28 (13)	43 (36)	32 (27)	5
kPAR	2377 (844)	992 (383)	269 (134)	132 (76)	879 (480)	331 (135)	10
RSRepair-A	208 (65)	103 (26)	34 (10)	↑ 62 (17)	250 (81)	62 (17)	2
jKali	92 (83)	83 (80)	14 (13)	7 (5)	35 (31)	30 (28)	3
Kali-A	43 (38)	4 (3)	8 (7)	2 (1)	12 (10)	3 (2)	3
Dynamoth	1 (0)	1 (0)	1 (0)	1 (0)	2 (1)	1 (0)	1
Nopol	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	2
ACS	15 (4)	11 (3)	2 (0)	2 (0)	15 (4)	↑ 17 (4)	17
Cardumen	966 (965)	141 (68)	87 (50)	77 (40)	479 (454)	77 (40)	2
ARJA	362 (302)	↑ 686 (648)	38 (22)	↑ 136 (116)	142 (117)	↑ 195 (184)	8
SimFix	3800 (3799)	2274 (2273)	166 (165)	↑ 307 (306)	1167 (1166)	745 (744)	24
FixMiner	546 (147)	357 (99)	111 (24)	77 (24)	754 (189)	609 (81)	13
AVATAR	1624 (512)	↑ 2426 (511)	164 (65)	136 (33)	478 (145)	↑ 530 (150)	19
TBar	2958 (1262)	1806 (1031)	240 (118)	120 (53)	818 (444)	620 (306)	24

*The upper whisker value is determined by 1.5 IQR (interquartile ranges) where IQR = 3rd Quartile - 1st Quartile, as defined in [9]. “All” denotes all fixed bugs, and “Correct” denotes correctly fixed bugs. The numbers outside the parentheses indicate the related NPC score values and the numbers inside the parentheses indicate the related $NPC_{in-plausible}$ score values. ↑ implies that the NPC and $NPC_{in-plausible}$ values of “Correct” are higher than those of “All”. “# bugs” denotes the number of bugs correctly fixed by each repair tool.

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 are checked.

Table 7: CR after setting a $NPC_{in-plausible}$ threshold.

Tool	Threshold*	# fixed bugs	CR(%)	Tool	Threshold*	# fixed bugs	CR(%)
jGenProg	80	3 (14)	+6.4	Nopol	0	2 (31)	0
GenProg-A	80	2 (25)	+1.3	ACS	32	17 (22)	0
jMutRepair	70	5 (20)	+2.3	Cardumen	70	2 (7)	+11.9
kPAR	400	10 (45)	+6.3	ARJA	650	8 (56)	+0.5
RSRepair-A	26	2 (27)	+2.5	SimFix	3800	24 (61)	+4.0
jKali	80	3 (22)	+1.6	FixMiner	620	13 (31)	+1.5
Kali-A	3	3 (26)	+6.9	AVATAR	511	19 (55)	+0.2
Dynamoth	0	1 (21)	+0.2	TBar	1230	24 (66)	+5.6

*The threshold for each repair tools is set with its upper-bound $NPC_{in-plausible}$ score shown in Figure 8.

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.

Overall our systematic study of patch generation efficiency reveals that (1) efficiency is not yet a widely-valued performance target; (2) state-of-the-art can avoid generating nonsensical patches; (3) the more templates an APR system considers, the more nonsensical and in-plausible patches it will generate; (4) specialized templates increase APR tool efficiency; (5) correct patches are sparse in the search space.

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 repairability (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 ACS, 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 anecdotical 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 patch following rule R2 (cf. Section 3.3.3). This finding reminds us that the benchmark that is used is not a complete ground-truth, neither for repair-oriented fault localization nor for patch generation.
- [*Fix location is different from bug location.*] We observe that jKali and Nopol also now fail to correctly fix respectively 2 and 1 bugs when they are given the developer-provided fix locations. This finding suggests that some repair tools are rather misled, in the cases of specific bugs, when they are 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.

Table 8: Impact† on repairability* when ground-truth fix locations are directly given to the APR system.

APR Tool	Chart	Closure	Lang	Math	Mockito	Time	Total	CR (%)
jGenProg	+1 (-3)	+1 (0)	0 (-2)	+2 (+1)	0 (0)	0 (0)	+4 (-4)	+28.8
GenProg-A	0 (-2)	+2 (+1)	+1 (+2)	+2 (-2)	0 (0)	0 (0)	+5 (-1)	+17.4
jMutRepair	0 (-3)	0 (-1)	0 (-2)	0 (-5)	0 (0)	0 (0)	0 (-11)	+22.8
kPAR	+2 (-5)	+5 (+11)	0 (-5)	+2 (-6)	0 (0)	+3 (+4)	+16 (0)	+25.4
RSRepair-A	0 (-2)	+2 (-6)	0 (+1)	+5 (0)	0 (0)	0 (0)	+7 (-7)	+21.6
jKali	0 (-3)	+1 (-6)	-1 (-4)	-1 (-4)	0 (0)	0 (0)	-1 (-17)	+13
Kali-A	0 (-5)	+2 (-18)	+1 (+3)	0 (-2)	0 (-1)	0 (0)	+3 (-23)	+9.7
DynaMoth	0 (-5)	N/A	+2 (+2)	0 (-5)	0 (0)	0 (-1)	+2 (-9)	+18.6
Nopol	-1 (-5)	N/A	0 (-3)	+1 (-13)	0 (0)	0 (-1)	0 (-22)	+15.7
ACS	0 (0)	0 (0)	-1 (-1)	0 (0)	0 (0)	0 (0)	-1 (-1)	**-1.1**
Cardumen	+2 (+2)	0 (-2)	0 (+1)	+1 (+3)	0 (0)	0 (0)	+3 (+4)	+14.6
AJRA	0 (-8)	+2 (-13)	-1 (+2)	+2 (-2)	0 (-1)	0 (0)	+3 (-18)	+16.8
SimFix	+2 (-4)	0 (-2)	0 (-10)	+2 (-4)	0 (0)	0 (0)	+5 (-18)	+22.7
FixMiner	+2 (-5)	+6 (+13)	+3 (+7)	+6 (+10)	+2 (+2)	+3 (+3)	+22 (+30)	+16.2
AVATAR	+1 (-4)	+3 (-2)	+1 (-2)	+4 (-4)	+2 (+2)	+2 (+3)	+13 (-7)	+30.6
TBar	+4 (-3)	+11 (+12)	+4 (-3)	+5 (-1)	+3 (+3)	+3 (+5)	+30 (+13)	+32.7

†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*

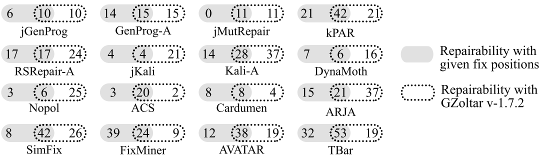

Figure 9: Overlap and difference between normal fault localization and given fix positions for repair tools.

- [*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 4, 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 [54] 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 is 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.

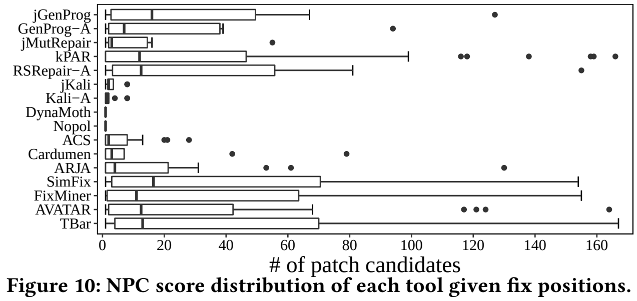

Figure 10: NPC score distribution of each tool given fix positions.

Fault localization is an important step in a repair pipeline. Its false positives, however, have a significant impact on both repairability and repair efficiency. In particular, we found that accurately localizing the bug can reduce the number of generated patches by an order of magnitude, thus drastically enhancing efficiency. From the perspective of repairability, better fault localization will increase the probability to generate correct patches (i.e., the correctness ratio).

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).

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, 55]. However, valid patches are sometimes incorrect as they overfit on incomplete test suites [48], and might cause issues during maintainance [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, 56, 62]. 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, serval studies simply reported the repair efficiency in terms of CPU time consumption of fixing bugs [12, 14, 55, 56, 62]. 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 in perspective against other tools. Our study fills this gap.

Empirical Study. To boost the development of program repair, various empirical studies have been conducted in this direction. Le et al. [24] re-assessed GenProg on real bugs, while several studies on overfitting followed [20, 23, 47, 48, 61]. Yang et al. [64] explored better test cases for better program repair. Yi et al. [65] empirically investigated the effectiveness of test-suite metrics in controlling the repairing reliability of GenProg. Motwani et al. [43] investigated to what extent important bugs can be fixed by 9 APR tools. Liu et al. [29] investigated the FL bias in benchmarking APR tools with only one APR tool. Durieux et al. [7] conducted a large-scale empirical study for Java APR tools to investigate their repairability on different benchmarks. Empirical studies for APR tools have been studied from different scenarios in the literature, but these studies mainly focus on the traditional APR tools and the latest state-of-the-art tools (e.g., ACS [62], SimFix [15] and TBar [31]) have not been studied systematically. Our study fills this gap by looking back at 10 years of test-based program repair research and focusing on the under-valued performance criterion that is efficiency.

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/APRStudy/APRStudy.git

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4 (July 2018), 81:1–81:37. https://doi.org/10.1145/3212695

[3] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A Learning-to-Rank Based Fault Localization Approach Using Likely Invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 177–188. https://doi.org/10.1145/2931037.2931049

[4] Liushan Chen, Yu Pei, and Carlo A Furia. 2017. Contract-based program repair without the contracts. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 637–647.

[5] Vidroha Debroy and W Eric Wong. 2010. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation*. IEEE, 65–74.

[6] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2017. Dynamic patch generation for null pointer exceptions using metaprogramming. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 349–358.

[7] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. *arXiv preprint arXiv:1905.11973* (2019).

[8] Thomas Durieux and Martin Monperrus. 2016. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the IEEE/ACM 11th International Workshop in Automation of Software Test*. IEEE, 85–91.

[9] Michael Frigge, David C Hoaglin, and Boris Iglewicz. 1989. Some implementations of the boxplot. *The American Statistician* 43, 1 (1989), 50–54.

[10] Zachary P. Fry, Bryan Landau, and Westley Weimer. 2012. A Human Study of Patch Maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 177–187. https://doi.org/10.1145/04000800.2336775

[11] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2017), 34–67.

[12] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 19–30.

[13] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*. AAAI, 1345–1351.

[14] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 12–23.

[15] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 298–309.

[16] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 437–440.

[17] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 802–811.

[18] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 946–957.

[19] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2018. Fixminer: Mining relevant fix patterns for automated program repair. *arXiv preprint arXiv:1810.01791* (2018).

[20] Xuan-Bach D Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On reliability of patch correctness assessment. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 524–535.

[21] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 593–604.

[22] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 213–224.

[23] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. *Empirical Software Engineering* 23, 5 (2018), 3007–3033.

[24] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE, 3–13.

[25] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[26] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* (2019).

[27] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, 55–56.

[28] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Shin Yoo, and Yves Le Traon. 2018. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering* (2018).

[29] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation*. IEEE.

[30] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 456–467.

[31] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 31–42.

[32] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. 2018. LSRepair: Live search of fix ingredients for automated program repair. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference ERA Track*.

[33] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 118–129.

[34] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*. IEEE, 702–713.

[35] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Vol. 51. ACM, 298–312.

[36] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 468–478.

[37] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether One of Two Random Variables Is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (March 1947), 50–60. https://doi.org/10.1214/aoms/1177730491

[38] Matias Martinez and Martin Monperrus. 2016. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 441–444.

[39] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: the Cardumen Mode of Astor. In *International Symposium on Search Based Software Engineering*. Springer, 65–86.

[40] Martin Monperrus. 2009. The living review on automated program repair. In *Proceedings of the Symposium on the Foundations of Software Engineering*. HAL/archives-ouvertes. fr, 315–324.

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

[42] Martin Monperrus. 2018. Automatic software repair: A bibliography. *Comput. Surveys* 51, 1 (2018), 17:1–17:24.

[43] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. 2018. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering* 23, 5 (2018), 2901–2947.

[44] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 772–781.

[45] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 609–620. https://doi.org/10.1109/ICSE.2017.62

[46] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 254–265.

[47] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. 2013. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *Proceedings of the 22nd International Symposium on Software Testing and Analysis*.

ACM, 191–201.

[48] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 24–36.

[49] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. 2018. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th IEEE/ACM International Conference on Mining Software Repositories*. IEEE, 10–13.

[50] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. Elixir: Effective object-oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 648–659.

[51] Seemanta Saha, Ripon K Saha, and Mukul R Prasad. 2019. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 13–24.

[52] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 532–543.

[53] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 130–140.

[54] Shangwen Wang, Ming Wen, Deheng Yang, and Xiaoguang Mao. 2018. Attention Please: Consider Mockito when Evaluating Newly Released Automated Program Repair Techniques. *arXiv preprint arXiv:1812.05370* (2018).

[55] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE, 364–374.

[56] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1–11.

[57] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exploring and Exploiting the Correlations Between Bug-Inducing and Bug-Fixing Commits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, New York, NY, USA, 326–337. https://doi.org/10.1145/3338906.3338962

[58] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 479–490.

[59] F. Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.

[60] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 660–670.

[61] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 789–799.

[62] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*. IEEE, 416–426.

[63] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55.

[64] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 831–841.

[65] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2018. A correlation study between automated program repair and test-suite metrics. *Empirical Software Engineering* 23, 5 (2018), 2948–2979.

[66] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering* (2018).