A Critical Review on the Evaluation of Automated Program Repair Systems

Kui Liu[a], Li Li[b,*], Anil Koyuncu[c], Dongsun Kim[d,*], Zhe Liu[a], Jacques Klein[c], Tegawendé F. Bissyandé[c]

[a]*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China*
[b]*Faculty of Information Technology, Monash University, Australia*
[c]*Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg*
[d]*School of Computer Science and Engineering, Kyungpook National University, South Korea*

Abstract

Automated Program Repair (APR) has attracted significant attention from software engineering research and practice communities in the last decade. Several teams have recorded promising performance in fixing real bugs and there is a race in the literature to fix as many bugs as possible from established benchmarks. Gradually, repair performance of APR tools in the literature has gone from being evaluated with a metric on the number of generated plausible patches to the number of correct patches. This evolution is necessary after a study highlighting the overfitting issue in test suite-based automatic patch generation. Simultaneously, some researchers are also insisting on providing time cost in the repair scenario as a metric for comparing state-of-the-art systems.

In this paper, we discuss how the latest evaluation metrics of APR systems could be biased. Since design decisions (both in approach and evaluation setup) are not always fully disclosed, the impact on repair performance is unknown and computed metrics are often misleading. To reduce notable biases of design decisions in program repair approaches, we conduct a critical review on the

*Corresponding authors.
Email addresses: `kui.liu@nuaa.edu.cn` (Kui Liu), `li.li@monash.edu` (Li Li), `anil.koyuncu@uni.lu` (Anil Koyuncu), `darkrsw@knu.ac.kr` (Dongsun Kim), `zhe.liu@nuaa.edu.cn` (Zhe Liu), `jacques.klein@uni.lu` (Jacques Klein), `tegawende.bissyande@uni.lu` (Tegawendé F. Bissyandé)

evaluation of patch generation systems and propose eight evaluation metrics for fairly assessing the performance of APR tools. Eventually, we show with experimental data on 11 baseline program repair systems that the proposed metrics allow to highlight some caveats in the literature. We expect wide adoption of these metrics in the community to contribute to boosting the development of practical, and reliably performable program repair tools.

Keywords: Automated program repair, evaluation, assessment, metrics.

1. Introduction

In the last decade, Automated Program Repair (APR) [1, 2] has extensively grown as the prominent research topic in the software engineering community. APR approaches aims to alleviate the manual effort involved in fixing software bugs. Recent approaches in the literature have achieved promising performance by highlighting the possibility to fix more and more real benchmark bugs automatically. As advocated by the research community, APR holds several promises: in production, it will drastically reduce the time-to-fix delays and limit downtime; in a development cycle, APR can help suggest changes to accelerate debugging. For the former, repair bots have started to show their power in the context of continuous integration bots [3]. For the latter, Facebook has recently reported on the traction that their Getafix [4] fix suggestion tool is getting among developers. In the literature, there are two distinct repair scenarios: (1) fixing *syntactic errors*, i.e., cases where the developer code violates some programming language specifications [5, 6] and (2) fixing *semantic bugs*, i.e., cases where the developer implementation of program behaviour deviates from developer's intention [7, 8]. The latter is the scope of the recent race in program repair. We will focus our review on the evaluation of such systems.

Since the work of Weimer et al. [9] ten years ago, the assessment of APR approaches in the literature attempts to provide information on the number of bugs for which APR tool can generate a patch that makes the buggy program pass all the test cases [9, 10, 11, 12, 13, 14, 15]. Six years after this seminal work on generate-and-validate patch generation systems, Qi et al.[16] and Smith et al. [17] have concurrently presented empirical results showing that generated patches can be *plausible* only: they can make the programs pass all the test cases but may not actually fix the bug, due to overfitting on the test suite. Therefore, researchers have started to focus some effort in automating the identification of patch correctness [18, 19]. Eventually, to fairly assess the performance on fixing real bugs of APR tools, the number of bugs for which a *correct* (i.e., it is semantically equivalent to the patch that the program developer accepts for fixing the bug) patch is generated appeared to be a more reasonable metric than the mere number of plausible patches [20]. This metric has since then become standard among researchers, and is now widely accepted in the literature for evaluating APR tools [21, 14, 15, 22, 23, 24, 25, 26, 27]. Table 1 provides an example of assessment results excerpted from the paper describing Hercules [28], one of the most recent state-of-the-art works on APR that was tested on the Defects4J [29] programs. Based on data reported in this table, researchers explicitly rank the APR systems, and use this ranking as a validation of new achievements in program repair.

Table 1: Table excerpted from [28] with the caption "*Statistics of Patch Generation by Various Techniques (Correct/Incorrect)*".

Subject	Math	Lang	Time	Chart	Closure	Total
Hercules	21/7	10/3	3/2	6/3	6/2	46/17
SimFix	14/12	9/4	1/0	4/4	6/2	34/22
CapGen	13/-	5/-	0/-	4/-	0/-	22/-
JAID	1/-	1/-	0/-	2/-	5/-	9/-
Elixir	12/7	8/4	2/1	4/3	0/-	26/15
ssFix	10/16	5/7	0/4	3/4	2/9	20/40
ACS	12/4	3/1	1/0	2/0	0/-	18/5

Unfortunately, various research experiences in developing and assessing APR tools have proven that "this comparison is non-trivial, and could further be largely biased due to a non-consideration of important details" [30] regarding *approach* and *evaluation* design decisions. *Approach design decisions* concern the technical implementation details such as the fault localization configuration in the APR workflow. *Evaluation design decisions* include non-implementation choices such as the execution platform and time out settings. Indeed, recent studies have suggested that an APR technique can achieve different performance on fixing bugs when reported executions are replicated under different configuration settings. For instance, in a recent study, Liu et al. [30] have demonstrated how repair performance comparison in the literature is biased by the use of diverse fault localization techniques. Nevertheless, the authors of [30] did not provide metrics that the community should use to improve fairness in comparison.

Besides the number of bugs that an APR approach can fix, which estimates effectiveness, APR assessment must consider measuring efficiency of the patch validation process, highlighting performance in terms of nature, complexity and importance of bugs, as well as clarifying to what extent the reported performance is likely reachable in practitioner settings. The objective of the paper is thus to propose metrics that limit the biases when assessing APR tools and when discussing comparison results.

Similar to the "overfitting" and "fault localization bias" studies, which helped to improve the assessment criteria of APR tools, our work aims at highlighting other potential biases in approach and evaluation design decisions when comparing different APR tools performance. We go beyond prior works by proposing a diverse set of metrics which limit comparison biases.

Our contributions in this paper are twofold:

1. We conduct a critical review on the evaluation of automated program repair systems. This review is based on experimental data collected by replicating the execution of 11 APR tools in the literature.

2. We propose bias-limited metrics to be used in conjunction with the widely-

accepted metric "number of bugs fixed with correct/plausible patches" for evaluating the repair performance of APR systems. These metrics attempt to reduce the comparison biases that we exhibit in our critical review:

- ***Upper bound Repair Performance*** metric aims to clearly provide an indication of the patch generation limitations when focusing on this part of the APR system (i.e., APR systems are given with the exact bug-fixing positions obtained from the ground-truth developers' patches).
- ***Fault Localization Sensitiveness*** metric aims to assess the impact of the used fault localization on the repair performance of the APR system.
- ***Patch Generation Efficiency*** metrics aim to clarify the APR efficiency, the effort to yield a plausible/correct patch.
- ***Bug Diversity*** metrics aim at evaluating APR system performance from intrinsic attributes of bugs.
- ***Benchmark Overfitting*** metrics aim to clarify the difference of APR systems performance between in-the-lab and in-the-wild assessment settings.

2. Background & Motivation

Automated program repair workflow, for generate-and-validate systems, includes three basic processes as shown in Figure 1: **fault localization** (FL), which produces a ranked list of suspicious code locations that must be modified to fix the bug, **patch generation** (PG), which implements the change operators that are applied on the code locations, and **patch validation** (PV), which executes the test cases to ensure that the patched program meets the expected behaviour encoded in the test suite. Each of these steps is obviously important and may significantly impact repair performance of generate-and-validate APR systems. For example, if FL code locations are majoritarily false positives, the APR system may generate patches that do never pass test cases. Liu et al. [30]

have even recently shown that current FL techniques are not all equivalent when comparing APR tools to assess their repair performance.

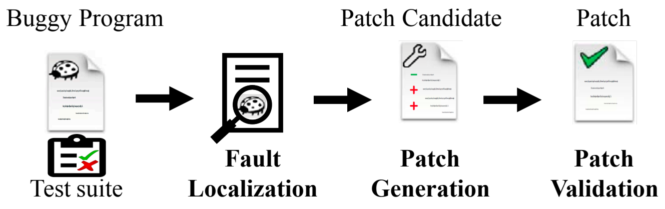

Figure 1: Standard steps in a pipeline of Automated Program Repair.

The current literature trend in APR studies is to evaluates the performance of APR systems based on the number of successfully fixed bugs [15, 31], which determines whether a generated patch is successful by counting the number of passing test cases. If a patch can pass all the given test cases (both previously-passing and previously-failing test cases on the buggy version), it is regarded as a successful patch. This criterion was first used by Weimer et al. [9] in their seminal work on generate-and-validate APR systems.

Unfortunately, as later studies have revealed, even if a generated patch can pass all test cases, it might break a necessary behavior or introduce other faults, which are not covered by the given test suite [17]. Moreover, a developer may not accept the patch due to several reasons such as coding convention [10, 32]. All such patches are often called **plausible patches** since they require further investigations to ensure that they are **correct patches** that would be 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.

Similarly, efficiency of APR tools is being assessed by researchers via measuring time to generate-and-validate patches. For example, very recently, Ghanbari et al. use time cost criterion to assess the efficiency of their APR tool (PraPR) [33] on real bugs. Table 2 excerpts the average time costs reported by Ghanbari et al. [33] for the different Defects4J subjects. On average, for each `Closure` bug, PraPR generated and validated 29,849.9 patches. This represents

10 times more patches than the ones generated and validated by PraPR for each `Chart` bug. Yet, the time cost for `Closure` bugs is 20 times more the time cost for `Chart` bugs.

Table 2: Table excerpted from [33] with the caption "*Average PraPR time cost (s)*".

Subjects	# Patches	Time cost (s)
Chart	2,827.6	157.8
Closure	29,849.9	3,027.3
Lang	544.4	210.2
Math	3,333.2	1,629.2
Mockito	2,601.0	148.8
Time	2,968.2	144.2

When considering the information shown in Table 3 about the Defects4J dataset, the ratios of line of source code (LoC), number of test cases, line of test code and number of assert statements among the subjects, we question their correlation with the PraPR time cost for validating patches. Eventually, we postulate that it could be biased to assess the efficiency of APR tools by measuring the average time cost for fixing a diverse set of program bugs. This bias is further exacerbated by the non-linear correlations with execution platform configuration settings for RAM, CPU, etc.

Additionally, it should be noted that different evaluation scenarios set different time constraints to their approaches for generating a valid patch that can make the patched buggy program pass all given tests successfully. Such constraints constitute biases when comparing APR tools. To sum up, considering the limitation of the current assessment criteria for APR systems in the literature, it is necessary to introduce solid evaluation metrics for enabling fair comparisons among APR systems, and thus re-focus the advancement of APR techniques on aspects beyond superficial enumeration of bugs that are being fixed under different settings. We propose such metrics following a critical review of the evaluation of patch generation systems, taking into account both

Table 3: **Defects4J benchmark information.**

Subjects	**# bugs**	**KLoC**	**# test cases**	**Test KLoC**	**# assert**
Chart	26	96	2,205	50	9,121
Closure	133	90	7,927	83	8,936
Lang	65	22	2,245	6	13,117
Math	106	85	3,602	19	9,512
Mockito	38	11	1,457	20	1,882
Time	27	28	4,130	53	17,658

*Data shown in this table are excerpted from Defects4J paper [29] and [34].

qualitative and quantitative concerns.

3. Critical Review of Design Decisions

Monperrus maintains a living review of program repair approaches and tools in the software engineering community [1]. This review is however focused on discussing the novel heuristics and techniques applied for patch generations, and further summarizes empirical findings. We follow this review to systematically identify all relevant Java APR works (listed in Table 4). We then propose to perform a review of assessment methodologies in the related literature. We focus in this work on generate-and-validate approaches, which are fairly popular in the program repair community. In this study, our metric definitions follow the summaries of the findings which we have made: most APR systems follow the same classical procedures for evaluating APR performance, which is mainly focused the raw numbers of plausibly/correctly fixed bugs and the execution costs. Our review explores various potential biases that could be overlooked by researchers when evaluating APR approaches. We mainly stress on the impact of these biases on the reported performance and propose adequate metrics to help in reporting results that would help cross-comparison for various approaches. To that end, our review investigates the following research questions:

- **RQ1**: *Which biases may carry the approach design decisions in terms of repair performance assessment?* With this research question we consider how to reduce the performance comparison biases that are due to technical workflow implementation differences, notably in the fault localization step.

- **RQ2**: *Which biases may carry the evaluation setups in terms of repair efficiency measurement?* With this research question we consider how to reduce the performance biases caused by diverse execution configurations (e.g., time limitation setting) and the environment (e.g., execution platform) for assessing effectiveness of APR systems?

- **RQ3**: *Which biases are overlooked when the performance comparison ignores the nature of the bugs?* With this research question we consider how to encourage meaningful advancements in APR by highlighting the need for deep comparison of repair performance based on actual nature of bugs that are addressed.

- **RQ4**: *Which biases do the current benchmarks carry?* With this research question, we highlight how the construction of the benchmark may mislead the interpretation of the recorded performance when considering an APR real-world usage potential.

Table 4: List of the reviewed APR tools in this study.

jGenProg [11], jKali [11], jMutRepair [11], HDRepair [12], Nopol [13], ELIXIR [14], JAID [21], ssFix [35], CapGen [22], SketchFix [23], ACS [20], SimFix [15], SOFix [25], LSRepair [24], kPAR [30], FixMiner [31], AVATAR [26], TBar [27], PraPR [33], ARJA [36],Hercules [28], VFix [37].

In the remainder of this section, experimental data on the metrics are provided for 11 APR systems from the literature: jGenProg, jMutRepair, jKali, Nopol, ARJA, ACS, SimFix, kPAR, FixMiner, AVATAR and TBar, since their source code is publicly available, and they are executable with a few modifications on their fault localization settings. Our experiments on repairing Java programs are based on the Defects4J benchmark [29]. We use this benchmark since all considered tools have been assessed on it by their authors, thus offering a reference for checking that the results of tool replications are in line with results reported by the authors.

3.1. RQ1: APR Pipeline Implementation Biases

As introduced in the previous sections, the APR pipeline implementation can carry biases. We focus in this work on the fault localization step. Although Liu et al. [30] experimentally pointed out the issues that exist with current evaluation scenarios, they do not discuss the metrics that APR authors should compute towards enabling a fair comparison of repair performance. We investigate the first research question by defining two metrics that (1) evaluate the actual performance that a given APR's patch generation component can achieve when the exact bug-fixing positions are given; and (2) estimate the impact on repair performance of the false positives induced by the APR's implementation of a given fault localization technique. Note that, all data for this research question are from our previous empirical study [38] on the efficiency of APR tools.

3.1.1. ***Upper bound Repair Performance***

We evaluate the upper bound performance of an APR tool with an assumption: all exact bug-fixing positions are provided and given as input to the APR system in order to focus on assessing the effectiveness of the patch generation component. This metric eliminates the bias due to fault localization performance (which is generally undisclosed or overlooked in the literature). As demonstrated in a recent study [30], comparison tables (such as Table 1) reported in several research papers are misleading: a given APR tool may be outperforming all others because of an improved bug localization, and not due to the detailed sophisticated heuristics proposed for patch generation.

The Upper bound Repair Performance (UbRP) metric is computed as the number of bugs for which a valid patch can be generated. This metric can be computed with either plausible or correct patches:

$$UbRP : A \to \mathbb{Z}^* \tag{1}$$

where A is a set of APR tools and $UbRP(apr \in A)$ gives a non-negative integer value ($\in \mathbb{Z}^*$) (i.e., # of correctly fixed bugs). Such metric is first proposed as the repair performance of APR tools with perfect fault localization in our previous study [30] and has been adopted in the community [26, 27, 39, 40, 41, 42].

This work is first to report the repair performance of 11 state-of-the-art APR tools in terms of *UbRP* scores, which are provided in Table 5 that lists the results by their publishing time. TBar outperforms the state-of-the-art tools by correctly fixing 54 bugs. Overall, the comparison data in Table 5 allow clarifying the advancements that *the Upper bound Repair Performance of APR tools has been significantly improved as time goes by.*

Table 5: Upper bound Repair Performance (i.e., # bugs that are correctly fixed when the exact bug-fixing positions are provided**).**

Subject	**Chart**	**Closure**	**Lang**	**Math**	**Mockito**	**Time**	**Total**
jGenProg	1	1	0	4	0	0	6
jMutRepair	1	2	0	2	0	0	5
jKali	0	2	0	0	0	0	2
Nopol	0	N/A	1	1	0	0	2
ACS	2	0	2	11	0	1	16
ARJA	1	4	1	5	0	0	11
kPAR	6	13	4	7	0	3	33
SimFix	4	7	5	12	0	1	29
FixMiner	7	6	4	12	2	3	34
AVATAR	6	9	5	6	2	2	30
TBar	10	15	10	13	3	3	54

3.1.2. ***Fault Localization Sensitiveness***

To fix bugs in the wild, APR systems rely on fault localization (FL) techniques to spot identify buggy code locations [30, 20, 13]. The accuracy of such localization can impact the overall repair performance since change operations are generally applied on (or around) the suspected code locations [30]. To estimate the impact of fault localization on APR tools within a benchmark dataset, we propose to compute a fault localization sensitiveness (Sen_{fl}) metric which takes as reference the correctness of patches with normal FL settings and the aforementioned *Upper bound Repair Performance* metric. Equation 2 defines the formula for computing the sensitiveness (Sen) metric:

$$\begin{aligned} &Sen : A \times FL \rightarrow [0,1] \\ &RP : A \times FL \rightarrow \mathbb{Z}^* \\ &Sen(apr \in A, fl \in FL) = \left(\frac{RP_{pn}(apr, fl)}{RP(apr, fl)} + \frac{UbRP'(apr)}{UbRP(apr)} \right) /2 \end{aligned} \tag{2}$$

where RP is the number of fixed bugs for the given APR ($apr \in A$) and FL tools. In RP, the number of correctly fixed bugs is denoted as RP_c, while RP_{pn} represents the number of plausibly fixed bugs by modifying the code on non-buggy positions. $UbRP'$ denotes the number of bugs that can be correctly fixed when APR tools are given with bug-fixing positions, but APR tools fail to correctly fix them because of the false positives of fault localization in the normal APR pipeline.

The Fault Localization Sensitiveness for an APR tool is therefore measured with the ratio of plausibly fixed bugs by modifying the code on non-buggy positions, and the percentage of bugs which could be correctly fixed when the exact bug positions are available but cannot be correctly fixed by the APR tool with its normal fault localization configuration. Table 6 provides data about the fault localization sensitiveness of 11 APR tools. If considering the fault localization sensitiveness, ACS outperforms state-of-the-art APR tools with 22.8 to 67.3 percentage points margin, which is followed by jMutRepair and SimFix.

There are two scenarios that may explain the good performance of ACS

Table 6: Fault localization sensitiveness of each APR tool.

Subjects	RP_{pn}	RP	$UbRP'$	$UbRP$	Sen
jGenProg	9	20	2	6	39.2%
jMutRepair	12	22	0	5	27.3%
jKali	13	25	1	2	51%
Nopol	29	31	1	2	71.8%
ACS	2	22	0	16	**4.5%**
ARJA	46	58	6	11	66.9%
kPAR	29	63	19	33	51.8%
SimFix	26	68	6	29	29.5%
FixMiner	16	33	22	34	56.6%
AVATAR	21	57	11	30	36.8%
TBar	27	72	29	54	45.6%

and SimFix. (1) Either its patch generation system is effective even in the presence of false positive buggy locations. Or (2) the fault localization technique used by ACS and SimFix is more effective. Our review of the work finds that ACS leverages the predicating switching [43] to improve the performance of fault localization, while SimFix uses a different version of GZoltar from other tools, and leverages test case purification [44] to improve the fault localization accuracy. Although both of them propose advanced technologies to search donor code for patch generation, it would be difficult to assess whether it is fault localization or the patch generation that makes ACS and SimFix less affected to potential noise in fault. Hence the importance of comparison APR tools with the *Upper bound Repair Performance* metric.

By comparing the state-of-the-art APR tools based on the "Upper bound repair Performance" and "Fault Localization Sensitiveness" metrics, we are able to highlight that the sophisticated patch generation heuristics that the authors described thoroughly cannot clearly take the credit on overall performance improvement.

Reducing or eliminating the bias caused by fault localization techniques is necessary to fairly assess the repair performance of APR systems. It would be fair to compare the performance on fixing real bugs for different APR tools with the same fault localization setting. However, it is impossible to ensure that all different APR tools would use the same fault localization technique. Even when different APR tools use the same fault localization framework, there are still various versions available. For example, GZoltar [45] is widely used in the state-of-the-art APR systems for Java bugs, but these APR systems did use the different released versions[1] of GZoltar. Therefore, it is difficult to eliminate the bias caused by various fault localization techniques. However, our proposed metrics can help estimate this bias by APR authors themselves.

3.2. RQ2: APR Efficiency Assessment Biases

The intention of researchers proposing APR systems is to contribute to reducing manual debugging effort by fixing bugs automatically. Therefore, if the patch generation is highly expensive in terms of time or computing resources, it may deter development teams. It is thus important to measure repair efficiency. Currently, this is widely assessed in the literature based on the time-to-validate generated patch candidates. This assessment is implicit when APR authors set time limitation constraints. Table 7 report time limitation settings for fixing a single bug as reported by different APR system evaluations in the literature: once a plausible patch is generated within the time limit, the repair process will stop this bug; otherwise the repair process will only abandon its attempt to fix

[1]`https://github.com/GZoltar/gzoltar/releases`

the bug when the time runs out.

Table 7: Time limitation settings of APR tools.

APR Tool	Time Limitation Setting	APR Tool	Time Limitation Setting
jGenProg [11]	3h	SimFix [15]	5h
jKali [11]	3h	SOFix [25]	3h
jMutRepair [11]	3h	LSRepair [24]	3h
HDRepair [12]	1.5h	kPAR [30]	3h
Nopol [13]	5h	FixMiner [31]	3h
ELIXIR [14]	1.5h	Avatar [26]	3h
JAID [21]	? >35h[†]	TBar [27]	3h
ssFix [35]	2h	PraPR [33]	? <1h[§]
CapGen [22]	1.5h	ARJA [36]	115h
SketchFix [23]	-[‡]	Hercules [28]	5h
ACS [20]	0.5h	VFix [37]	3h

*h: hour. The time limitation setting of JAID and SketchFix is not clearly specified in their papers.

[†]The time limitation of JAID must be bigger than 35 hours as it costs 2228.6 minutes to fix bug Lang-24 in Defects4J [21].

[‡]SketchFix does not set the time limitation since it fixes bugs by mutating the top 50 suspicious statements generated by the fault localization technique.

[§]PraPR does not clearly specify its time limitation setting which should be less than 1 hour, as it can validate 35,521 candidate patches within an hour [33].

Given the diversity in computing capacities of the different execution platforms, the time limitation setting for patch validation constitutes an undiscussed bias when present comparison results: for example, 3 hours may or may not be enough to validate all generated patches for a given bug depending on the platform. This bias is ignored to the point where the same researchers may select different time limitations for their different tools and still attempt to compare the overall repair performance metrics. For example, ACS [20] and SimFix [15], although they have been developed and studied by the same research group, are evaluated with different time limitation settings on different platforms and yet compared on the same equal ground (cf. Table 1).

In order to highlight the biases that time cost information carry, we run two simple experiments to measure the CPU run time occupied by the processes for compiling and for running all test cases for the Defects4J buggy programs. To that end, we use the standard commands provided in the Defects4J setup (i.e., "`defects4j compile`" and "`defects4j test`") on two different machines: Machine-1 (OS X El Capitan Version 10.11.6 with 2.5 GHz Intel Core i7, 16GB 1600MHz DDR3 RAM) and Machine-2 (macOS Mojave Version 10.14.1 with 2.9 GHz Intel Core i9, 32 GB 2400MHz DDR4 RAM).

As shown in Figure 2 and Figure 3, Machine-1 will take significantly[2] more time than Machine-2 to compile and test each buggy program irrespectively of the project. Additionally, the time cost for compiling and testing different bugs do not equate. Thus, time limitations could lead to biases in the assessment of repair performance by the same APR system for different bugs.

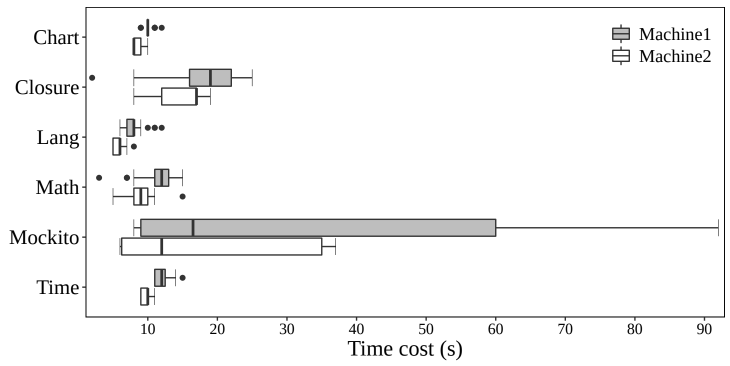

Figure 2: Time cost of compiling each Defects4J bug on two different machines.

Finally, Table 8 summarizes information on the variety of experimental platforms that can be found in the literature describing assessment results of several APR systems targeting Java program bugs. These information is often provided when reviewer report time cost details. Such a practice is prevalent in the systems community where all operating system details must be disclosed in the assessment of performance overhead. Unfortunately, while in the system pro-

[2]We have performed the MWW test to ensure that the difference of median values is statically significant for each distribution pair

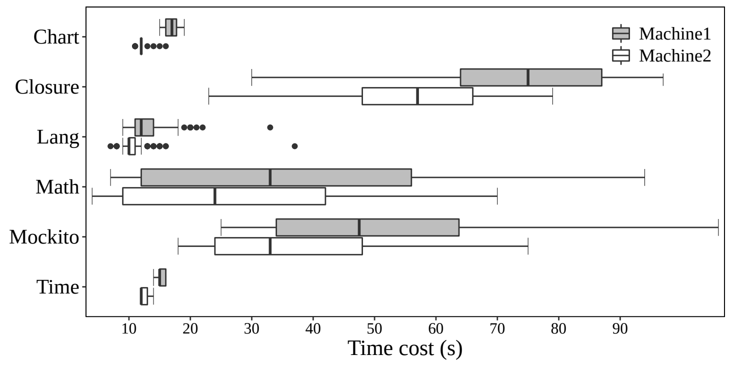

Figure 3: Time cost of testing each Defects4J bug on two different machines.

gramming community this overhead is computed in a fine-grained manner, focusing on the target functionality, in APR, several layers are considered together leading to potential biases.

Overall, overlooking platform configurations variations when implementing constraints in repair delay, or when reporting time-to-validate costs may lead to comparison biases. Thus, to limit this pervasive bias in the literature we propose a generic *APR Patch Generation Efficiency* metric which simply considers, for a given bug, the number of patch candidates (NPC)[3] generated by an APR system when the first valid patch is found. We recall that a generate patch may be of one of the following categories (w.r.t. to the buggy program):

1. **Nonsensical patch**. Such a patch cannot even make the patched buggy program successfully compile [10, 32].

2. **In-plausible patch**. Such a patch makes the patched buggy program successfully compile, but fails to pass some test cases in the available test suite.

3. **Valid patch**. Such a patch makes the patched program successfully pass

[3]The NPC score was initially proposed by Qi et al. [46] to measure the performance of fault localization techniques with APR tools.

Table 8: Platform information of running APR tools.

APR Tool	Platform Information
jGenProg	N/A
jKali	N/A
jMutRepair	N/A
HDRepair	a machine with one 2.4 GHz Intel Core i5-2435M CPU and 8GB memory.
Nopol	a PC with an Intel Core i7 3.60 GHz CPU and a Debian 7.5 operating system.
ELIXIR	an Ubuntu 14.04 LTS operating system with 2 Core of Intel(R) Core(TM) i7-4790 CPU of 3.60GHz and 4GB memory.
JAID	a cloud infrastructure with Ubuntu 14.04 system, one core of an Intel Xeon Processor E5-2630 v2, 8 GB of RAM
ssFix	a PC with 8 AMD Phenom(tm) II processors and 8G RAM.
CapGen	a CentOS server with 2x Intel Xeon E5-2450 Core CPU@2.1 GHz and 192GB physical memory.
SketchFix	a platform with 4-core Intel Core i7-6700 CPU (3.40 GHz) and 16 Gigabyte RAM on Ubuntu Linux 16.04.
ACS	an Ubuntu virtual machine with i7 4790K 4.0GHz CPU and 8G memory.
SimFix	a 64-bit Linux server with two Intel(R) Xeon CPUs and 128GB RAM. Each bug is assigned with 2 CPU cores and 8GB RAM.
SOFix	a Ubuntu server with 2.00 GHzIntel Xeon E5-2620 CPU and 16GBs of memory.
LSRepair	a HPC computing system with 24 Intel Xeon E5-2680 v3 cores with 2.GHz per core and 3TB RAM.
kPAR	a HPC computing system with 24 Intel Xeon E5-2680 v3 cores with 2.GHz per core and 3TB RAM.
FixMiner	a HPC computing system with 24 Intel Xeon E5-2680 v3 cores with 2.GHz per core and 3TB RAM.
Avatar	a HPC computing system with 24 Intel Xeon E5-2680 v3 cores with 2.GHz per core and 3TB RAM.
TBar	a HPC computing system with 24 Intel Xeon E5-2680 v3 cores with 2.GHz per core and 3TB RAM.
PraPR	a Dell workstation with Ubuntu16.04.4 LTS, IntelXeon CPU E5-2697 v4@2.30GHz and 98GB RAM.
ARJA	an Intel Xeon E5-2680 V4 2.4 GHz processor with 20 GB memory.
Hercules	a cluster of Ubuntu Virtual Machines with double core 3.6GHz processor and 4GB memory.
VFix	an Intel Core i5 3.20 GHz CPU and 4GB memory.

* "?" means the related platform information for running Astor [11] (including jGenProg, jKali and jMutRepair) is not clearly specified.

315 all test cases in the test suites [16]. A *correct patch* is a valid patch that actually fixes the bug [16] (i.e., is not simply overfitted to the test suite). *Correctness* is generally determined manually [20, 38, 26, 27, 15, 22].

The validation of *nonsensical patches* only costs the time to run compilation processes. Other patches are more expensive to validate since the patched program must be tested against all test cases in the test suite. Therefore, we refine the proposed NPC-score metric into two sub-metrics that compute the numbers of *nonsensical* and *in-plausible* patches that are generated before a valid patch. These are defined as:

$$\begin{aligned} &NPC : A \times B \rightarrow \mathbb{Z}^* \\ &N_{np} : A \times B \rightarrow \mathbb{Z}^* \\ &N_{ip} : A \times B \rightarrow \mathbb{Z}^* \end{aligned} \tag{3}$$

where NPC represents the number of patch candidates generated when the first valid patch is found by the given APR tool $apr \in A$ for the bug $b \in B$. 320 A and B represent a set of APR tools under consideration and a set of bugs fixed by the APR tools, respectively. While N_{np} and N_{ip} return the number of nonsensical/in-plausible patches when the first valid patch, respectively. $NPC = N_{np} + N_{ip} + N_v$, and N_v represents the first valid patch.

The repair efficiency of APR tools has been detailed present in our previous 325 study [38] in terms of the NPC scores. In this work, we further recall the importance of repair efficiency of APR tools with the average NPC scores that are not presented before. Table 9 reports the NPC, N_{np} and N_{ip} scores for 11 APR tools to correctly fix the bugs that they eventually manage to address. The APR tools that rely on fix patterns (i.e., SimFix, kPAR, FixMiner, AVATAR and 330 TBar) seem to generate more patch trials than other APR tools. While SimFix does not generate any nonsensical patches for each bug: the employed heuristics for donor code search appears to be effective. However, in the absence of such a metric, the authors could not better highlight this aspect in their evaluation.

Repair efficiency comparisons can be biased by APR execution platform settings (e.g., time limitation settings or computing capability details machines). Computing the number of eventually-useless patches that must be validated before a valid patch patch is found stands as a platform-independent and reliable metric for comparison.

Table 9: Average NPC, N_{np} and N_{ip} scores when the first valid patch is found.

Subject	N_{np}	N_{ip}	NPC
jGenProg	16/175	20/493	37/669
jMutRepair	1/3	9/31	11/35
jKali	0/6	2/36	3/43
Nopol	0	0	1/1
ACS	3/10	3/5	7/16
ARJA	4/24	26/117	31/142
kPAR	31/389	39/481	71/871
SimFix	0	284/1167	285/1168
FixMiner	117/564	136/190	254/755
AVATAR	37/333	16/145	54/479
TBar	41/374	35/445	77/820

*In each column, we provide x/y numbers: x is the number of generated patches when buggy code positions are provided for APR tools; y is the number of generated patches with the normal fault localization setting. **Note** that the APR tools presented in this Table can be re-executed successfully that are different from Tables 7 and 8.

3.3. RQ3: APR Bug and Patch Diversity Biases

After reviewing recent APR assessment studies in the literature, we note that the evaluation of repair performance of APR systems is mainly based on effectiveness (in the form of numbers of fixed bugs) and efficiency (in terms of time costs) comparisons. Although, we have proposed metrics to relieve these assessments from related biases, we further note that the intrinsic attributes (e.g., category, complexity or priority) of bugs is not considered when comparing the performance of different approaches.

For example, some APR tools target specific bugs: Nopol [13] and ACS [20] have been proposed to address conditional statements-related bugs, NPEfix [47] is focused on null pointer exception bugs while Genesis targets fixing three classes of defects (namely, null pointer, out of bounds, and class cast) [48]. On the other hand, several state-of-the-art APR systems do not target any specific bug type. Consequently, comparing focused APR systems with broad APR systems could lead to biased conclusions. Instead, as Monperrus [32] claimed since 2014, the classes of bugs that can be addressed by an APR system may also reflect its repair performance.

In this section, we present the assessing metrics for APR systems from intrinsic attributes of bugs and discuss the importance of evaluating APR systems with them. Note that, to eliminate the differences of repairing results between re-executing the APRs and their original executions caused by fault localization bias [30], in this section, the data about the correctly fixed bugs for each APR tool are directly excerpted from their published research papers.

3.3.1. **Effectiveness in Applying Diverse Repair Patterns**

The literature reports several studies [26, 49, 50] on mining fix patterns to implement state-of-the-art program repair tools. Eventually, the APR purpose should be to include several repair patterns in order to potentially fix a large set of bugs. In this section, we consider the three APR tools (i.e., kPAR, TBar and SimFix) as examples to present the differences of fixing effectiveness in applying diverse repair patterns. The three APR tools are selected with the following reasons: 1) kPAR [30] is the Java implementation of PAR [10] that is the first-proposed fix pattern based APR tool, 2) TBar[27] summarizes the fix patterns released in the literature, and 3) SimFix [15] is a heuristic-based APR tools with code change patterns (i.e., fix patterns) and achieves the fixing performant outperforming the state-of-the-art APR tools. For example, the fix pattern "`Replacement (Method Invocation, Method Invocation)`" in SimFix presents replacing the buggy method invocation with the correct one [15], which is illustrated as "`Method Replace`" pattern in PAR [10] and "`Mutate Method Invocation Expression`" pattern in TBar [27]. This fix pattern is consistent with the repair pattern "`Wrong Method Reference` (wrongMethod)" in the dissertation work of Defects4J bugs [51].

Consider the common comparison scenario depicted in Figure 4: such a Venn diagram is typically used in the literature [15] to highlight the repair achievement reached by a given state-of-the-art technique in terms of the number of bugs that it can exclusively fix compared to its competitors.

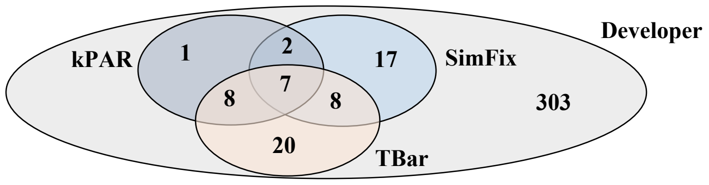

Figure 4: Overlaps in # *of Bugs* that are correctly fixed by kPAR, SimFix and TBar. *Developer* represents the benchmark universe of developer-fixed bugs.

380 Based on the displayed data, it seems that SimFix and TBar are largely disjoint in the bugs that they can repair successfully. Nevertheless this comparison can be biased given that the reason behind this limited overlapping may not be related to the bug types themselves but rather on extra-factors already reported in previous sections. For example, although two bugs are of the same 385 types (e.g., Null pointer exception), an APR tool may validate an overfitted patch for one of them and generate a correct one for the other.

To limit the bias in overlooking the nature of the bugs, we propose as an effectiveness metric to count the number of repair actions that the APR tool covers. To that end, we consider the repair actions as enumerated by Sobreira et al. [51] in their dissection study of Defects4J bugs and patches. Indeed, we approximate in this metric the nature of bugs with the properties of the patches that are applied on them. The following equation represents this metric:

$$N_{ra} : A \times PT \rightarrow \mathbb{Z}^* \tag{4}$$

where PT is a set of repair patterns defined by [51] and $N_{ra}(apr \in A, pt \in PT)$ returns the number of repair actions belonging to pt, which is implemented by apr.

Figure 5 presents the Venn diagram highlighting the overlapping in the application of diverse repair actions. The displayed data now shows that kPAR, SimFix and TBar are largely similar in the implemented repair operations. The adoption of such a metric will call for more qualifications of the over-performance reported by APR authors.

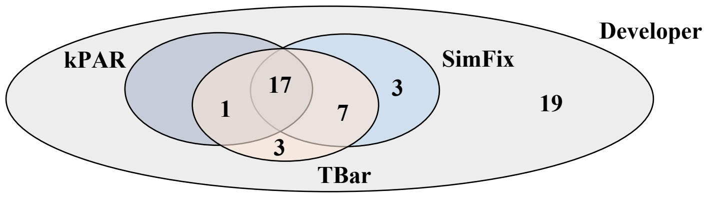

Figure 5: Overlaps in # of *Repair actions* that are used by kPAR, SimFix and TBar to correctly fixed bugs. *Developer* represents the repair actions used for all developer-provided benchmark patches.

Eventually, we provide in Figure 6 the distribution of repair patterns[4] used by TBar, SimFix and kPAR for the different bugs in the benchmark. With such a representation, it is easy to note which patterns each APR tool is able to successfully use as operator for generating correct patches: although human developers use the same patterns as APR tools, they are able to fix significantly more bugs with these patterns, raising the acute issue for APR to find the appropriate fix ingredients (i.e., donor code problem). For example, fixing a wrong method reference (i.e., *wrongMethodRef*) requires identifying the right method to be used instead. While developers can readily identify such a method, APR tools are still experimenting with heuristics to find method candidates either within the same file (TBar) or in the same project (SimFix).

3.3.2. Effectiveness in Addressing more or less Complex Bugs

Besides the nature of the bugs that are fixed, APR evaluation should qualify the complexity of the correctly fixed bugs. Complexity however is a relative concept depending on the subjects. In this work, we propose to simply approx-

[4]following the dissection of Defects4J patches by Sobreira et al. [51]

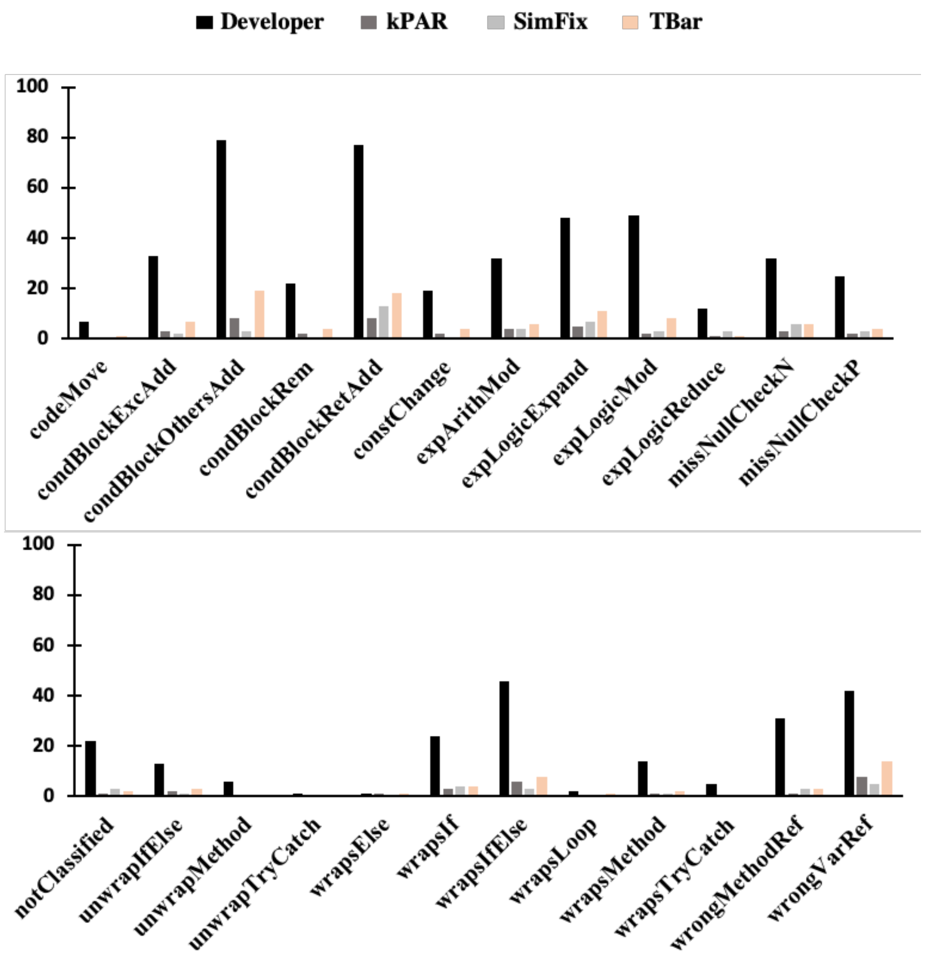

Figure 6: Repair patterns implemented by Developers (i.e., Benchmark patches), kPAR, TBar and SimFix. The x-axes represent repair pattern taxonomies excerpted from is from [51]. The y-axes represent the number of Defects4J bugs in taxonomies.

imate complexity with the *number of code statements that are impacted by the APR change operations to correctly fix a bug*, which is also used to assess the importance of APR-fixed bugs in the literature [52]. It should be noted that, in practice, a statement might span across several lines while the code in one line might encompass several statements (depending on code formatting conventions). We further define the "*impacted statements*" as the statements that are used to fix a bug in terms of modifying the code with one of four code change actions (i.e., "`Update`", "`Delete`", "`Insert`", and "`Move`") [53, 54]. This metric can be defined as:

$$N_{stmt} : A \times \mathbb{N} \to \mathbb{Z}^* \tag{5}$$

where $N_{stmt}(apr, n)$ takes an APR tool ($apr \in A$) and a number of impacted statements ($n \in \mathbb{N}$. e.g., $n = 1, 2$ or more), and gives the number of successful patches (e.g., zero or positive integer values) generated by the tool.

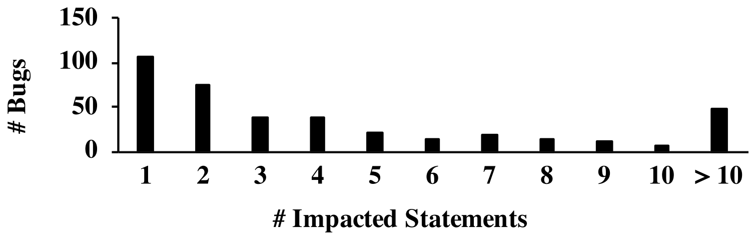

Figure 7: Complexity Distribution of 395 Defects4J Bugs.

410 Figure 7 illustrates the distribution of numbers of impacted statements (identified with a code differencing tool, GumTree [53]) for the 395 Defects4J bugs by considering the developer patches. Almost half of the bugs can be fixed by mutating only 1 or 2 statements. Nevertheless, APR tools should attempt to cover bugs of varying complexity. In table 10, we report complexity details of
415 the bugs that are correctly fixed by 22 APR tools, respectively. All the 22 APR systems can mainly fix bugs with low complexity (i.e., with one, two or three statements impacted).

3.3.3. **Effectiveness in Addressing more or less Important Bugs**

In addition to diversity and complexity of fixed bugs, we investigate the *importance* of the bugs that APR tools manage to correctly fix. We define *importance* as the extent to which the bug prevents correct functioning of the program. We estimate it via the program functionalities that are broken due to bug. In this study we approximate functionalities with test cases available in the test suite. Indeed, as postulated by Weimar et al. [9], test cases encode

Table 10: Complexity of Defects4J Bugs Fixed by APR tools.

# of impacted statements	1	2	3	4	5	6	7	8	9	10	>10
# bugs	107	74	39	39	21	15	19	15	11	6	49
jGenProg	2	2	1	0	0	0	0	0	0	0	0
jKali	0	0	1	0	0	0	0	0	0	0	0
jMutRepair	3	0	0	0	0	0	0	0	0	0	0
HDRepair	3	2	1	0	0	0	0	0	0	0	0
Nopol	2	1	1	1	0	0	0	0	0	0	0
ELIXIR	24	1	1	0	0	0	0	0	0	0	0
JAID	19	4	2	0	0	0	0	0	0	0	0
ssFix	16	2	1	0	0	0	0	1	0	0	0
CapGen	20	1	0	0	0	0	0	0	0	0	0
SketchFix	17	0	2	0	0	0	0	0	0	0	0
ACS	5	6	1	4	0	0	1	1	0	0	0
SimFix	20	8	2	0	0	0	1	1	1	0	1
SOFix	22	0	1	0	0	0	0	0	0	0	0
LSRepair	11	3	3	1	0	0	0	0	1	0	0
kPAR	15	1	2	0	0	0	0	0	0	0	0
FixMiner	23	2	0	0	0	0	0	0	0	0	0
AVATAR	17	1	2	3	0	0	1	3	0	0	0
TBar	29	4	2	3	0	0	1	3	0	0	1
PraPR	37	1	3	0	0	0	1	1	0	0	0
ARJA	5	10	1	1	0	1	0	0	0	0	0
Hercules	30	11	1	5	0	0	1	1	0	0	0
VFix	4	2	2	2	0	0	0	1	0	0	1

the functionality requirements of the program (i.e., the expected behaviours). If a bug makes the program fail to pass more or less test cases than others, we can estimate that it should have more or less priority to be fixed. This metric defines as:

$$N_{test} : A \times \mathbb{N} \to \mathbb{Z}^* \tag{6}$$

where $N_{test}(apr, n)$ returns the number of successful patches generated by an APR tool ($apr \in A$) for bugs with n failing test cases.

Table 11 presents the distribution of failing test cases for the bugs that are correctly fixed by 22 APR tools. Most Defects4J bugs indeed concern a single test case. However, the data shows that current APR tools are not performing well on bugs that concern several test cases. While the state-of-the-art SimFix approach, which implements sophisticated heuristics for fix ingredient search, manages to fix significantly more complex bugs than the kPAR baseline (cf. Table 10), these bugs are not of more importance (i.e., they relate to a single test case) than those fixed by the baseline APR tool. This may suggest that the applied change operators are still focused on solving single functional issues.

Repair effectiveness measurements can be biased if APR assessment overlooks the nature, complexity and importance of the bugs that different tools can address. Comparison among the state-of-the-art tools should strive to clarify the bug properties in order to highlight the power of the implemented repair patterns.

3.4. RQ4: APR Benchmark Biases

Although generate-and-validate APR research has been initiated a decade ago, the current benchmarks have been proposed mid-way: Defects4J [29], the widely used benchmark for Java bugs was released in 2014, while IntroClass and ManyBugs benchmarks [55] for C program bugs were published in 2015. The former is a collection of real-world project bugs, while the latter are collected from students' programs development as part of their curriculum work which reduces confidence on their relevance as real-world bugs.

Given the limitation of publicly available and reliable benchmarks of real bugs for evaluating APR systems, several APR systems may be overfitting to the available benchmarks, therefore lacking generalizability on program bug targets. Nevertheless, benchmark providers must put more effort to make their dataset readily usable (cf. discussion on Section 4) to attract APR researchers.

Table 11: Importance of fixed Defects4J bugs - Importance is approximated by the Number of Failing Program Test Cases.

# of failed test cases	1	2	3	4	5	6	7	8	9	10	>10
# bugs	251	72	23	12	5	5	7	8	2	2	8
jGenProg	5	0	0	0	0	0	0	0	0	0	0
jKali	1	0	0	0	0	0	0	0	0	0	0
jMutRepair	3	0	0	0	0	0	0	0	0	0	0
HDRepair	5	1	0	0	0	0	0	0	0	0	0
Nopol	5	0	0	0	0	0	0	0	0	0	0
ELIXIR	25	0	0	0	0	0	0	0	0	0	1
JAID	19	4	0	0	1	0	0	0	0	0	1
ssFix	18	0	1	0	0	0	1	0	0	0	0
CapGen	20	0	0	0	0	0	0	0	0	0	1
SketchFix	16	2	1	0	0	0	0	0	0	0	0
ACS	13	3	0	2	0	0	0	0	0	0	0
SimFix	24	7	1	1	0	0	1	0	0	0	0
SOFix	21	0	0	0	0	0	0	0	0	0	2
LSRepair	18	0	0	0	0	0	0	0	0	0	1
kPAR	13	4	0	0	0	0	0	0	0	0	1
FixMiner	19	3	0	0	0	0	0	0	0	0	3
AVATAR	14	8	1	1	0	0	1	0	0	0	2
TBar	26	11	1	2	0	0	1	0	0	0	2
PraPR	31	6	2	0	1	0	1	0	0	0	2
ARJA	14	4	0	0	0	0	0	0	0	0	0
Hercules	34	7	2	2	0	1	1	1	0	0	1
VFix	3	3	0	2	0	0	0	1	0	0	3

Bias with Processed bugs. By investigating the Defects4J dataset, which is widely used in the Java APR literature, we note that the benchmark authors have taken steps to curate the buggy programs in a state where fault localization and program repair tools can be applied. In this study, we summarize the curation of test suites into three categories: (1) to include the bug-triggering test cases which are often added after a user-reported bug is fixed by a developer, (2) to rewrite previous test cases to focus on the spotted buggy lines, and (3) to insert assertions in program source code in order to ensure that test cases properly reveal the bugs. Eventually, Defects4J is a clean research dataset which may not represent the types of buggy programs APR tools will be applied on in real-world development settings [56].

Table 12 provides details on the number of benchmark bugs in Defects4J [29], Bugs.jar [57] and Bears [58] that are associated with bug-triggering test cases that have been added/updated after the bug is reported. For such bugs, it seems a test generation process would have been necessary before any classical generate-and-validate APR tool can be applied. We thus propose as final metrics that APR assessment should report the *Numbers of correctly fixed bugs among the benchmark subsets of "unprocessed" and "processed" bugs.* For the implementation of this metric, *"processed"* and *"unprocessed"* bugs are real-world buggy programs for which benchmark authors did or did not perform any curation task, respectively. Curation task in this study is evaluated through the future test cases (i.e., bug-triggering test cases) that are added or updated with aforementioned three curation categories, as such test cases were not available at the time of the bug is reported and have thus been inserted into the benchmark dataset by their authors. The metrics are defined as follows:

$$N_{nfut} : A \to \mathbb{Z}^* \tag{7}$$

$$N_{fut} : A \to \mathbb{Z}^* \tag{8}$$

where $N_{nfut}(apr)$ and $N_{fut}(apr)$ return the number of successful patches generated by an APR tool ($apr \in A$) when future test cases are provided or not, respectively.

For example, 381 of Defects4J bugs are patched and validated with future test cases. We have performed experiments where we dropped such test cases from the test suites to align with practitioner settings: <u>kPAR, SimFix and TBar, with each its normal fault localization technique, can only fix 3, 2 and 6 bugs</u>

Table 12: **Future Test Cases in Benchmark Dataset.**

Benchmark	# Bugs	# bugs integrated with future test cases
Defects4J	395	381
Bugs.jar*	1,130 (1,151)	1,064
Bears	251	232

*Bugs.jar contains 21 duplicated bugs which are already contained in the remaining 1,130 bugs.

with correct patches, respectively. This finding suggests that the state-of-the-art may have not yet improved the applicability of program repair to real-world practitioner settings.

Although APR benchmarks are often built from real-world project data, many bugs are actually processed leading to a bias in validating that APR tools are ready for production environments. APR assessment should therefore explicitly differentiate experiments that are valid in-the-lab from those that would approximate in-the-wild performance. This can be done by dividing the considering subsets of processed vs unprocessed bug artefacts (i.e., buggy program code and available test cases).

4. Discussion

4.1. Benchmark Overfitting

While Defects4J has become a de-facto benchmark for Java APR, the generalizability of APR tools beyond its bugs must be questioned. Fortunately, in recent years, more benchmark datasets are being built and released for the community. Saha et al. [57] collected 1,158 bugs from 8 Java open source projects to build benchmark `bugs.jar`. Madeiral et al. [58] collected 251 bugs from 72 Java open source projects to build benchmark `Bears`, the largest benchmark of reproducible Java program bugs with respect to project diversity (the closest benchmark is Defects4J and Bugs.jar, which cover only six and eight projects, respectively). For C programs, two new benchmarks, DBGBench [59] and Codeflaws [60], are built by collecting bugs from real-world programs.

While preparing the experiments for our study, we attempted to use the Bugs.jar and Bears benchmark. Unfortunately, we failed to evaluate three APR systems on them despite committing resources for adapting the pipeline of the three APR systems to such benchmarks. There are mainly two reasons for our failure:

1. It is difficult to compile all bugs in the two benchmarks with a single version of JDK and maven. The bugs have indeed been collected from different versions of programs that are configured with different incompatible versions. This limits the usability of the whole benchmark as researchers must assess each bug independently on different execution environments.

2. Because the studied APR systems were specifically implemented for assessment with Defects4J, many pipeline scripts are mapped on Defects4J configurations. Thus, it is difficult to reuse other benchmarks without an invasive reorganisation of the APR toolkit code.

In contrast, Defects4J provides a friendly interface for its users for compiling and running test suites in a single execution environment. To date, Defects4J has been widely used in the test and repair community for Java programs, while Bugs.jar, which is much larger and more comprehensive in terms of artefacts (e.g., bug reports) was only used by the benchmark authors themselves for their APR assessment [14]. Eventually, the community must work together on building diverse and readily-usable benchmarks.

4.2. Findings and Future Work

The computed metrics for kPAR, SimFix, and TBar have revealed a number of avenues for APR research towards improving performance. As previously presented in Figure 5, 15 repair actions that are necessary to fix some Defects4J bugs are neither implemented in the SimFix state-of-the-art nor in the TBar and kPAR baseline APR systems. Future research on APR should consider investigating the addition of such associated change operations in their patch generation systems.

We further note that despite being able to apply the required repair actions, SimFix, TBar, and kPAR still fail to fix 303 bugs. This is indicative that, beyond the repair actions, APR research needs to further focus on improving the patch generator ability to search for corner code, or to efficiently rank patch candidates.

4.3. Threats to Validity

Although the proposed metrics are generic, our study on their importance carries some threats to validity. First, as a threat to external validity, we only considered three APR systems targetting Java program bugs. Nevertheless, we minimized this threat by considering a variety of criteria and under the constraint that many APR systems proposed in the literature are not reproducible (e.g., some are not even open source): we consider early and recent APR systems (spanning seven years of research). One is a state-of-the-art (with sophisticated heuristics) while others are baselines (with naive pattern match and transform engines). One implements few fix patterns while another is more comprehensive.

As a threat to internal validity, we have rerun the experiments by reconfiguring the setup scripts of the different APR tools so that they work on our execution platforms. We minimize the associated threat by ensuring that our results match the ones reported by the authors in their original papers.

As threats to construct validity, we have used patch properties as proxies to qualify bugs. Actually, we could have used bug report details. Nevertheless, the Defects4J benchmark bug reports were limited, which would make the findings limited subsequently.

Finally, a general threat to validity is that the metrics, which compute numbers, maybe too generic. For APR assessment, each produced metric needs to have a qualitative discussion. For example, with respect to the complexity metric, we have investigated the case of TBar, which manages to correctly fix some bugs by applying change operations to multiple statements (i.e., >two statements). By carefully checking such bugs, we have noted that all of them are fixed by simply deleting the buggy statements. Thus, the metric by itself is not comprehensively explaining the performance of the APR system. It is the sum of all metrics that provide an overview of reliable performance comparisons.

5. Related Work

5.1. Evaluation Criteria in APR

While it is widely used in the APR literature [9, 61, 62, 8], simply counting the number of fixed bugs cannot accurately assess the actual effectiveness of APR tools since a generated patch could be nonsensical [10, 32] even if it can pass all the given test cases. Fundamentally, test cases can reflect specific aspects of a program so that passing all the given test cases may not imply to satisfy the required functionality of the program. The nonsensical patch would have negative impacts on software maintainability as well [63]. This issue in evaluating APR tools is called 'overfitting' [17, 32]. After this discussion, most studies report both the numbers of plausible (fixed but overfitting with test cases) and correct patches.

In most APR studies, the upper bound of time cost has been arbitrarily specified while it could be one of the key performance criteria as we discussed in Section 3.2. Some APR tools based on evolutionary algorithms [9, 62] used the number of generations that the tools can take at most. Other studies employ a wall-clock time bound, (e.g., running an APR tool at most three hours and assuming the tool cannot find a working patch until the time bound). Determining the time bound is currently a rule of thumb and experiment environments vary with different studies (e.g., a study runs a tool on an in-house computing resource [9] or another study adopts cloud-computing resources [62]). Thus, it is necessary to create a baseline experimental environment and time bound criterion in the community.

5.2. Criticism on Evaluation Methods in Software Engineering

As many novel techniques have been proposed in the literature of software engineering, several researchers tried to rethink the current evaluation method is correct to assess existing techniques. While the APR community actively publish critical reviews [32, 64], other disciplines also pay attention to the correctness of evaluation methods. Campos et al. [65] pointed out that the choice of evolutionary algorithm has not been properly evaluated in test suite generation and empirically evaluated existing techniques on 13 different algorithms. Lee et al. [66] criticized that the current practice of assessing IRBL (Information Retrieval-based Bug Localization) tools is biased since the authors often use outdated subjects and incorrect experiment setup. Razzaq et al. [67] reviewed feature localization techniques (FLT) and pointed out that most FLT studies did not clearly specify the experimental setup. Thus, the authors suggested guidelines of empirical evaluation methods for FLT.

6. Conclusion

Numbers of Plausible/Correct patches and time costs have been used to evaluate the repair performance of APR systems. However, since design decisions (both in approach and evaluation setup) are rarely fully disclosed in the assessment description, their impact on repair performance is overlooked, leading to misleading comparison results. Through a critical review of Java APR literature, we identify notable biases of design decisions and evaluation settings in program repair assessment. We then propose eight evaluation metrics for fairly assessing the performance of APR tools. Eventually, we show with experimental data on two baseline program repair systems as well as on a state-of-the-art APR system that the proposed metrics allow to highlighting some caveats in the literature. We expect wide adoption of the proposed metrics to contribute to boosting the development of practical, and reliably performing program repair tools.

Acknowledgements

This work is supported by the Project 1015-YAH20102, the National Natural Science Foundation of China (Grant No.61802180), the Natural Science Foundation of Jiangsu Province (Grant No.BK20180421), the National Cryptography Development Fund (Grant No.MMJJ20180105) and the Fundamental Research Funds for the Central Universities (Grant No.NE2018106). This work is also partially supported by the Fonds National de la Recherche (FNR), Luxembourg, under projects RECOM- MEND 15/IS/10449467 and FIXPATTERN C15/IS/9964569.

References

[1] M. Monperrus, The living review on automated program repair, in: Proceedings of the Symposium on The Foundations of Software Engineering, HAL/archives-ouvertes. fr, 2009, pp. 315–324.

[2] M. Monperrus, Automatic software repair: A bibliography, ACM Computing Surveys 51 (1) (2018) 17:1–17:24. `doi:10.1145/3105906`.

[3] S. Urli, Z. Yu, L. Seinturier, M. Monperrus, How to design a program repair bot?: insights from the repairnator project, in: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ACM, 2018, pp. 95–104. `doi:10.1145/3183519.3183540`.

[4] A. Scott, J. Bader, S. Chandra, Getafix: Learning to fix bugs automatically, Proceedings of the ACM Programming Languages 3 (OOPSLA) (2019) 159:1–159:27. `doi:10.1145/3360585`.

[5] R. Gupta, S. Pal, A. Kanade, S. Shevade, DeepFix: Fixing common c language errors by deep learning, in: Proceedings of the 31st AAAI Conference on Artificial Intelligence, AAAI Press, 2017, pp. 1345–1351. `doi:10.5555/3298239.3298436`.

[6] S. Bhatia, R. Singh, Automated correction for syntax errors in programming assignments using recurrent neural networks, arXiv preprint arXiv:1603.06129.

[7] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, A. Roychoudhury, Semantic program repair using a reference implementation, in: Proceedings of the 40th International Conference on Software Engineering, ACM, 2018, pp. 298–309. `doi:10.1145/3180155.3180247`.

[8] H. D. T. Nguyen, D. Qi, A. Roychoudhury, S. Chandra, SemFix: Program repair via semantic analysis, in: Proceedings of the 35th International Conference on Software Engineering, IEEE, 2013, pp. 772–781. `doi:10.1109/ICSE.2013.6606623`.

[9] W. Weimer, T. Nguyen, C. Le Goues, S. Forrest, Automatically finding patches using genetic programming, in: Proceedings of the 31st International Conference on Software Engineering, IEEE, 2009, pp. 364–374. `doi:10.1109/ICSE.2009.5070536`.

[10] D. Kim, J. Nam, J. Song, S. Kim, Automatic patch generation learned from human-written patches, in: Proceedings of the 35th International Conference on Software Engineering, IEEE, 2013, pp. 802–811. `doi:10.1109/ICSE.2013.6606626`.

[11] M. Martinez, M. Monperrus, ASTOR: a program repair library for java (demo), in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, 2016, pp. 441–444. `doi:10.1145/2931037.2948705`.

[12] X. B. D. Le, D. Lo, C. Le Goues, History driven program repair, in: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering, IEEE, 2016, pp. 213–224. `doi:10.1109/SANER.2016.76`.

[13] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, M. Monperrus, Nopol: Automatic repair of conditional statement bugs in java programs, IEEE Transactions on Software Engineering 43 (1) (2017) 34–55. doi:10.1109/TSE.2016.2560811.

[14] R. K. Saha, Y. Lyu, H. Yoshida, M. R. Prasad, ELIXIR: effective object-oriented program repair, in: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2017, pp. 648–659. doi:10.1109/ASE.2017.8115675.

[15] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, X. Chen, Shaping program repair space with existing patches and similar code, in: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, 2018, pp. 298–309. doi:10.1145/3213846.3213871.

[16] Z. Qi, F. Long, S. Achour, M. Rinard, 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, 2015, pp. 24–36. doi:10.1145/2771783.2771791.

[17] E. K. Smith, E. T. Barr, C. Le Goues, Y. Brun, Is the cure worse than the disease? overfitting in automated program repair, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015, pp. 532–543. doi:10.1145/2786805.2786825.

[18] Y. Xiong, X. Liu, M. Zeng, L. Zhang, G. Huang, Identifying patch correctness in test-based program repair, in: Proceedings of the 40th International Conference on Software Engineering, ACM, 2018, pp. 789–799. doi:10.1145/3180155.3180182.

[19] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, T. F. Bissyandé, Evaluating representation learning of code changes for predicting patch correctness in program repair, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2020. doi:10.1145/3324884.3416532.

[20] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, L. Zhang, Precise condition synthesis for program repair, in: Proceedings of the 39th IEEE/ACM International Conference on Software Engineering, IEEE, 2017, pp. 416–426. doi:10.1109/ICSE.2017.45.

[21] L. Chen, Y. Pei, C. A. Furia, Contract-based program repair without the contracts, in: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2017, pp. 637–647. doi:10.1109/ASE.2017.8115674.

[22] M. Wen, J. Chen, R. Wu, D. Hao, S.-C. Cheung, Context-aware patch generation for better automated program repair, in: Proceedings of the 40th International Conference on Software Engineering, ACM, 2018, pp. 1–11. doi:10.1145/3180155.3180233.

[23] J. Hua, M. Zhang, K. Wang, S. Khurshid, Towards practical program repair with on-demand candidate generation, in: Proceedings of the 40th International Conference on Software Engineering, ACM, 2018, pp. 12–23. doi:10.1145/3180155.3180245.

[24] K. Liu, A. Koyuncu, K. Kim, D. Kim, T. F. Bissyandé, LSRepair: Live search of fix ingredients for automated program repair, in: Proceedings of the 25th Asia-Pacific Software Engineering Conference ERA Track, 2018, pp. 658–662. doi:10.1109/APSEC.2018.00085.

[25] X. Liu, H. Zhong, Mining stackoverflow for program repair, in: Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering, IEEE, 2018, pp. 118–129. doi:10.1109/SANER.2018.8330202.

[26] K. Liu, A. Koyuncu, D. Kim, T. F. Bissyandé, 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, 2019, pp. 456–467. doi:10.1109/SANER.2019.8667970.

[27] K. Liu, A. Koyuncu, D. Kim, T. F. Bissyandé, TBar: Revisiting template-based automated program repair, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019, pp. 31–42. `doi:10.1145/3293882.3330577`.

[28] S. Saha, R. K. Saha, M. R. Prasad, Harnessing evolution for multi-hunk program repair, in: Proceedings of the 41st IEEE/ACM International Conference on Software Engineering, IEEE, 2019, pp. 13–24. `doi:10.1109/ICSE.2019.00020`.

[29] R. Just, D. Jalali, M. D. Ernst, Defects4j: A database of existing faults to enable controlled testing studies for java programs, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ACM, 2014, pp. 437–440. `doi:10.1145/2610384.2628055`.

[30] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, Y. L. Traon, 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, 2019, pp. 102–113. `doi:10.1109/ICST.2019.00020`.

[31] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, Y. L. Traon, FixMiner: Mining relevant fix patterns for automated program repair, Empirical Software Engineering 25 (3) (2020) 1980–2024. `doi:10.1007/s10664-019-09780-z`.

[32] M. Monperrus, 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, 2014, pp. 234–242. `doi:10.1145/2568225.2568324`.

[33] A. Ghanbari, S. Benton, L. Zhang, Practical program repair via bytecode mutation, in: Proceedings of the 28th ACM SIGSOFT Interna-

tional Symposium on Software Testing and Analysis, 2019, pp. 19–30. doi:10.1145/3293882.3330559.

[34] R. Just, C. Parnin, I. Drosos, M. D. Ernst, Comparing developer-provided to user-provided tests for fault localization and automated program repair, in: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, 2018, pp. 287–297. doi:10.1145/3213846.3213870.

[35] Q. Xin, S. P. Reiss, Leveraging syntax-related code for automated program repair, in: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2017, pp. 660–670. doi:10.1109/ASE.2017.8115676.

[36] Y. Yuan, W. Banzhaf, ARJA: Automated repair of java programs via multi-objective genetic programming, IEEE Transactions on Software Engineering (2018) 1–1doi:10.1109/TSE.2018.2874648.

[37] X. Xu, Y. Sui, H. Yan, J. Xue, VFix: value-flow-guided precise program repair for null pointer dereferences, in: Proceedings of the 41st International Conference on Software Engineering, IEEE, 2019, pp. 512–523. doi:10.1109/ICSE.2019.00063.

[38] K. Liu, S. Wang, A. Koyuncu, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, Y. Le Traon, On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs, in: Proceedings of the 42nd International Conference on Software Engineering, ACM, 2020, pp. 615–627. doi:10.1145/3377811.3380338.

[39] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, L. Tan, CoCoNuT: Combining context-aware neural translation models using ensemble for program repair, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, 2020, pp. 101–114. doi:10.1145/3395363.3397369.

[40] L. Chen, Y. Pei, C. A. Furia, Contract-based program repair without the contracts: An extended study, IEEE Transactions on Software Engineering (2020) 1–1doi:10.1109/TSE.2020.2970009.

[41] T. Lutellier, L. Pang, H. V. Pham, M. Wei, L. Tan, ENCORE: ensemble learning using convolution neural machine translation for automatic program repair, arXiv preprint arXiv:1906.08691.

[42] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, M. Monperrus, SequenceR: Sequence-to-sequence learning for end-to-end program repair, IEEE Transactions on Software Engineering (2019) 1–1doi:10.1109/TSE.2019.2940179.

[43] X. Zhang, N. Gupta, R. Gupta, Locating faults through automated predicate switching, in: Proceedings of the 28th International Conference on Software Engineering, ACM, 2006, pp. 272–281. doi:10.1145/1134285.1134324.

[44] J. Xuan, M. Monperrus, Test case purification for improving fault localization, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2014, pp. 52–63. doi:10.1145/2635868.2635906.

[45] J. Campos, A. Riboira, A. Perez, R. Abreu, GZoltar: an eclipse plug-in for testing and debugging, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2012, pp. 378–381. doi:10.1145/2351676.2351752.

[46] Y. Qi, X. Mao, Y. Lei, C. Wang, Using automated program repair for evaluating the effectiveness of fault localization techniques, in: Proceedings of the 22nd International Symposium on Software Testing and Analysis, 2013, pp. 191–201. doi:10.1145/2483760.2483785.

[47] T. Durieux, B. Cornu, L. Seinturier, M. Monperrus, Dynamic patch generation for null pointer exceptions using metaprogramming, in: Proceed-

ings of the 24th International Conference on Software Analysis, Evolution and Reengineering, IEEE, 2017, pp. 349–358. doi:10.1109/SANER.2017.7884635.

[48] F. Long, P. Amidon, M. Rinard, Automatic inference of code transforms for patch generation, in: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ACM, 2017, pp. 727–739. doi:10.1145/3106237.3106253.

[49] R. Rolim, G. Soares, R. Gheyi, L. D'Antoni, Learning quick fixes from code repositories, arXiv preprint arXiv:1803.03806.

[50] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, Y. Le Traon, Mining fix patterns for findbugs violations, IEEE Transactions on Software Engineering (2018) 1–1doi:10.1109/TSE.2018.2884955.

[51] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, M. de Almeida Maia, 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, 2018, pp. 130–140. doi:10.1109/SANER.2018.8330203.

[52] M. Motwani, S. Sankaranarayanan, R. Just, Y. Brun, Do automated program repair techniques repair hard and important bugs?, Empirical Software Engineering 23 (5) (2018) 2901–2947. doi:10.1007/s10664-017-9550-0.

[53] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus, Fine-grained and accurate source code differencing, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ACM, 2014, pp. 313–324. doi:10.1145/2642937.2642982.

[54] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, Y. Le Traon, A closer look at real-world patches, in: Proceedings of the 34th IEEE International

Conference on Software Maintenance and Evolution, IEEE, 2018, pp. 275–286. doi:10.1109/ICSME.2018.00037.

[55] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, W. Weimer, The manybugs and introclass benchmarks for automated repair of c programs, IEEE Transactions on Software Engineering 41 (12) (2015) 1236–1256. doi:10.1109/TSE.2015.245451.

[56] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, Y. L. Traon, iFixR: bug report driven program repair, in: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, 2019, pp. 314–325. doi:10.1145/3338906.3338935.

[57] R. Saha, Y. Lyu, W. Lam, H. Yoshida, M. Prasad, 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, 2018, pp. 10–13. doi:10.1145/3196398.3196473.

[58] F. Madeiral, S. Urli, M. Maia, M. Monperrus, 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, IEEE, 2019, pp. 468–478. doi:10.1109/SANER.2019.8667991.

[59] M. Bohme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, A. Zeller, Where is the bug and how is it fixed? an experiment with practitioners, in: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ACM, 2017, pp. 117–128. doi:10.1145/3106237.3106255.

[60] S. H. Tan, J. Yi, Yulis, S. Mechtaev, A. Roychoudhury, Codeflaws: a programming competition benchmark for evaluating automated program repair tools, in: Proceedings of the 39th International Conference on Software Engineering Companion, IEEE Press, 2017, pp. 180–182. doi:10.1109/ICSE-C.2017.76.

[61] C. Le Goues, T. Nguyen, S. Forrest, W. Weimer, GenProg: A generic method for automatic software repair, IEEE Transactions on Software Engineering 38 (1) (2012) 54–72. doi:10.1109/TSE.2011.10.

[62] C. Le Goues, M. Dewey-Vogt, S. Forrest, W. Weimer, A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each, in: 2012 34th International Conference on Software Engineering, 2012, pp. 3 –13. doi:10.1109/ICSE.2012.6227211.

[63] Z. P. Fry, B. Landau, W. Weimer, 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, 2012, pp. 177–187. doi:10.1145/04000800.2336775.

[64] S. Wang, M. Wen, X. Mao, D. Yang, Attention please: Consider mockito when evaluating newly proposed automated program repair techniques, in: Proceedings of the Evaluation and Assessment on Software Engineering, EASE '19, ACM, New York, NY, USA, 2019, pp. 260–266. doi:10.1145/3319008.3319349.

[65] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, A. Arcuri, An empirical evaluation of evolutionary algorithms for unit test suite generation, Information and Software Technology 104 (2018) 207–235. doi:10.1016/j.infsof.2018.08.010.

[66] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, Y. Le Traon, Bench4bl reproducibility study on the performance of ir-based bug localization, in: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, ACM, New York, NY, USA, 2018, pp. 61–72. doi:10.1145/3213846.3213856.

[67] A. Razzaq, A. Wasala, C. Exton, J. Buckley, The state of empirical evaluation in static feature location, ACM Transactions on Software Engineering and Methodology 28 (1) (2018) 2:1–2:58. doi:10.1145/3280988.