You Cannot Fix What You Cannot Find!

An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems

Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg
{kui.liu, anil.koyuncu, tegawende.bissyande, dongsun.kim, jacques.klein, yves.letraon}@uni.lu

Abstract—**Properly benchmarking Automated Program Repair (APR) systems should contribute to the development and adoption of the research outputs by practitioners. To that end, the research community must ensure that it reaches significant milestones by reliably comparing state-of-the-art tools for a better understanding of their strengths and weaknesses. In this work, we identify and investigate a practical bias caused by the fault localization (FL) step in a repair pipeline. We propose to highlight the different fault localization configurations used in the literature, and their impact on APR systems when applied to the Defects4J benchmark. Then, we explore the performance variations that can be achieved by "tweaking" the FL step. Eventually, we expect to create a new momentum for (1) full disclosure of APR experimental procedures with respect to FL, (2) realistic expectations of repairing bugs in Defects4J, as well as (3) reliable performance comparison among the state-of-the-art APR systems, and against the baseline performance results of our thoroughly assessed `kPAR` repair tool. Our main findings include: (a) only a subset of Defects4J bugs can be currently localized by commonly-used FL techniques; (b) current practice of comparing state-of-the-art APR systems (i.e., counting the number of fixed bugs) is potentially misleading due to the bias of FL configurations; and (c) APR authors do not properly qualify their performance achievement with respect to the different tuning parameters implemented in APR systems.**

Index Terms—**Automated Program Repair, Spectrum-based Fault Localization, Benchmarking, Empirical Assessment, Bias.**

I. Introduction

Automated program repair (APR) holds the promise of reducing the manual debugging effort by automatically generating patches for defects identified in a program. In production, APR will drastically reduce the time-to-fix delays and limit downtime. In a development cycle, APR can help suggest changes to accelerate debugging. In the literature, there are two distinct repair scenarios: (1) fixing *syntactic errors*, i.e., cases where code violates some programming language specifications [1], [2] and (2) fixing *semantic bugs*, i.e., cases where implementation of program behaviour deviates from developer's intention [3], [4]. The latter requires Fault Localization (FL) through execution of test cases. It is thus the scope of this paper.

Once a symptom of a fault is detected, most of recent APR systems follow the same basic pipeline as shown in Figure 1: (1) localize the fault, (2) generate a candidate patch, and (3) validate the patch. The first step (fault localization) of an APR system identifies an entity in a program as the potential fault location. In the second step (patch generation), given a fault location, the APR system modifies the program, i.e., creates a patch. The last step (patch validation) assesses whether the patch actually fixes the defect. If the patch is not regarded as a valid patch, the second and last steps are repeated until a valid patch is generated or the termination condition is satisfied. To increase the chances of finding a valid patch, the process is iterated over all suspicious code locations ranked by FL tools.

In the repair pipeline, APR systems generally focus on the patch generation step, but tend to use similar strategies for fault localization and patch validation. To the best of our knowledge, most of the current state-of-the-art APR approaches [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20] leverage test suites to perform fault localization and patch validation. For fault localization, the systems rely on a testing framework such as GZoltar [21], and a spectrum-based fault localization formula [22], [23], [24], such as Ochiai [25]. Eventually, bug fixing performance is measured by counting the number of bugs for which the system can generate a patch that passes all test cases. Such patches are claimed to be valid.

Nevertheless, given the growing interest in APR among software engineers, it is important to ensure that the research outputs are relevant and well assessed in terms of reliable performance for practitioners. In this respect, the APR research community has already started to reflect on the *acceptability* [7], [26] and *correctness* [27], [28] of the patches generated by APR tools. Researchers [27], [29], [30], [31], [32] raised the concern of overfitting patches: those are generated patches that can pass the validating test cases, but may actually not be the semantically-correct patches for repairing the defect.

Since then, assessment of APR approaches in the literature attempts to provide information on the number of generated patches that are *plausible* (i.e., they make the programs pass all the test cases) and the number of patches that are *correct* (i.e., they are equivalent to the patches that were actually submitted

TABLE I: Table excerpted from [33] with the caption "*Correct patches generated by different techniques*".

Proj.	SimFix	jGP	jKali	Nopol	ACS	HDR	ssFix	ELIXIR	JAID
Chart	4	0	0	1	2	-(2)	3	4	2(4)
Closure	6	0	0	0	0	-(7)	2	0	5(9)
Math	14	5	1	1	12	-(7)	10	12	1/(7)
Lang	9	0	0	3	3	-(6)	5	8	1/(5)
Time	1	0	0	0	1	-(1)	0	2	0/(0)
Total	34	5	1	5	18	13(23)	20	26	9/(25)

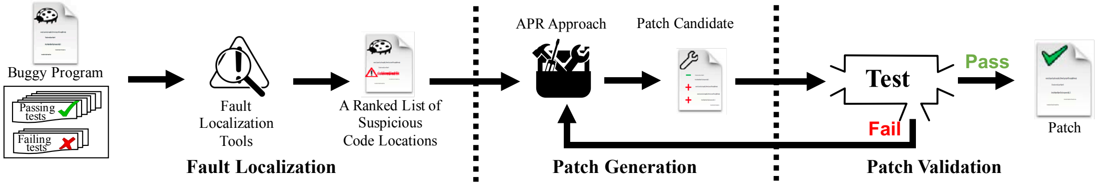


Fig. 1: Standard steps in a pipeline of Automated Program Repair.

by the program developers). Table I provides an example of assessment results excerpted from the paper describing SimFix [33], one of the most recent state-of-the-art works on APR that was tested on the Defects4J [34] 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.

Unfortunately, our own experience in developing and assessing APR tools has proven that this comparison is non-trivial, and could further be largely biased due to a non-consideration of important details regarding the FL step. Indeed, recall that an APR technique cannot attempt to generate relevant patch candidate unless the FL step can successfully identify the target buggy code locations in a program. Thus, FL accuracy across repair pipelines can impact, either by boosting or degrading, the performance of an APR system.

For example, SimFix [33] and ACS [19], although they have been developed by the same research group, are evaluated on different versions of a fault localization technique without discussing the impact of such a change in the experimental configuration. As another example bias, while most APR techniques simply integrate off-the-shelf fault localization tools in the repair pipelines, in some experiments, such as for HDRepair [13], whose authors make the assumption that the buggy method is known. Unfortunately, this assumption gives an important advantage as the list of suspicious code statements is limited and likely to include the buggy statement, thus leading to overestimation of the performance.

> *Similar to the "overfitting" study, which helped to improve the assessment criteria of APR tools, our work aims at highlighting the potential biases in comparing different APR approaches without any consideration of implementation variations of the FL step.*

Overall, our investigation into the relationship between fault localization performance and APR tool performance seeks to provide answers to the following research questions (RQs):

RQ1 ***How do APR systems leverage FL techniques?*** We first investigate FL techniques used in APR systems in the literature. This reveals which FL tool and formula are integrated for each APR system. We examine implementation details of each APR system, and/or directly ask the authors of the technique to clarify FL configuration, e.g., which level of detection granularity is considered, and how many suspicious locations are considered.

RQ2 ***How many bugs from a common APR benchmark are actually localizable?*** After aggregating APR performance data reported in the literature, we note that 246 bugs (in benchmark Defects4J) have not yet been fixed by any state-of-the-art APR tool. Given that researchers scarcely discuss the reasons behind repair misses, we assess, with this research question, our intuition that FL is possibly one of the challenging steps in the repair pipeline.

RQ3 ***To what extent APR performance can vary by adapting different FL configurations?*** We implement and make publicly available `kPAR`, a straightforward fix pattern-based APR system, and record its performance under various configurations to serve as a comparable baseline for future research.

Eventually, we make the following contributions:

- We expose a hidden bias throughout the comparison of APR tools in the literature, and present more reliable performance comparisons for current state-of-the-art.
- We build and make publicly available an easy-to-configure fault localization toolkit that can be adopted in APR pipelines for Java programs.
- We provide a refined benchmark for evaluating the performance of APR systems with respects to those bugs that can actually be localized.
- We implement and make publicly available a baseline APR system with its different performance metrics for different FL configurations.

Our replication package, including `kPAR`, is available at:

`https://github.com/flvsapr/FL-VS-APR`

II. Background

We recall how fault localization is important in an APR pipeline, and describe how current APR systems are assessed.

A. Fault Localization in Automated Program Repair

In APR systems, fault localization (FL) is not only the first step but also seriously affects the performance of the systems. Given a buggy program (with its passing and failing test cases), an FL tool is leveraged during the FL step to identify the suspicious buggy code locations as described in Figure 1. The granularity of suspicious locations can be a file, method, or line. Ideally, the location should be both precise and accurate. If the precision is low (e.g., the granularity is broad such as file), the patch generation step needs to explore a large space of candidate patches. If the accuracy is low (e.g., the FL step provides a wrong fault location), the subsequent step generates patches for the non-faulty program entity.

Spectrum-based fault localization (SBFL, also referred to as coverage-based fault localization) [22], [23], [24] is one of the most popular FL techniques used in APR systems. This technique applies a ranking metric to detect faulty code

TABLE II: Number of bugs reported having been fixed by different APR tools. *APR systems are ordered by year of publication.*

Proj.	**jGenProg** [35]	**jKali** [35]	**jMutRepair** [35]	**HDRepair** [13]	**Nopol** [18]	**ACS** [19]	**ELIXIR** [36]	**JAID** [15]	**ssFix** [37]	**CapGen** [38]	**SketchFix** [39]	**FixMiner** [40]	**LSRepair** [41]	**SimFix** [33]
Chart	0/7	0/6	1/4	0/2	1/6	2/2	4/7	2/4	3/7	4/4	6/8	5/8	3/8	4/8
Closure	0/0	0/0	0/0	0/7	0/0	0/0	0/0	5/11	2/11	0/0	3/5	5/5	0/0	6/8
Lang	0/0	0/0	0/1	2/6	3/7	3/4	8/12	1/8	5/12	5/5	3/4	2/3	8/14	9/13
Math	5/18	1/14	2/11	4/7	1/21	12/16	12/19	1/8	10/26	12/16	7/8	12/14	7/14	14/26
Mockito	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0
Time	0/2	0/2	0/1	0/1	0/1	1/1	2/3	0/0	0/4	0/0	0/1	1/1	0/0	1/1
Total	5/27	1/22	3/17	6/23	5/35	18/23	26/41	9/31	20/60	21/25	19/26	25/31	19/37	34/56
P(%)	18.52	4.55	17.65	26.09	14.29	78.26	63.41	29.03	33.33	84.00	73.08	80.65	51.35	60.71

† In each column, we provide x/y numbers: x is the number of correctly fixed bugs; y is the number of bugs for which a plausible patch is generated by the APR tool (i.e., a patch that makes the program pass all test cases). The same as other similar tables.

locations by leveraging the execution traces of test cases to calculate the likelihood (based on *suspiciousness scores*) of program entities to be faulty. The ranking metric is applied to calculate suspiciousness scores for program entities (such as program statements as well as code lines [42]).

In the APR literature [19], [33], [35], [39], [38], Ochiai [25] is widely used as the ranking metric of SBFL. Many empirical studies [43], [44], [23] have indeed shown that Ochiai is one of the most effective techniques in localizing the root cause of faults in object-oriented programs. Ochiai computes suspiciousness score of a given source code statement s following the formula of Equation 1:

$$S_{ochiai}(s) = \frac{failed(s)}{\sqrt{(failed(s) + passed(s)) * (failed(s) + failed(\neg s)}} \quad (1)$$

where $failed(s)$ and $passed(s)$ denote respectively the number of failing and passing tests that executed statement s, while $failed(\neg s)$ represents the number of failing test cases that do not execute statement s. In practice, FL tools eventually report a ranked list of statements associated with the suspiciousness scores.

B. APR Performance Assessment

The current practice of APR studies often evaluates the performance of APR systems based on the number of successfully fixed bugs [33], [40]. We can determine 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 passing and failing cases given for the buggy version), it is regarded as a successful patch.

However, the number of passing test cases may not correctly assess the effectiveness of generated patches. 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 [27]. Moreover, a developer may not accept the patch due to several reasons such as coding convention [7], [26]. These patches are often called **plausible patches** since it needs further investigations to check whether they are **correct patches** acceptable to developers. In the literature, *correctness* is assessed manually by comparing the generated against the developer-provided patch available in the benchmark.

Similarly, selecting a FL technique could be another issue since it can make the performanc assessment biased. Our investigations will use Table II as a starting point to highlight the problem of fault localization bias. This table shows the number of fixed bugs out of the bugs in the Defects4J [34] benchmark, which are reported by the authors of the current state-of-the-art APR tools in the literature. The results of jGenProg, jKali and Nopol are extracted from the experimental data reported by Martinez et al. [45]. The results of other tools are collected from data reported by papers' authors in the literature.

III. Experimental Setup

Our experiments are based on common tool-support and processes used in the literature. We clarify the experiment design in this section as the basis for understanding the implementation and the conclusions that we draw.

A. Definition of Fault Locality

Although state-of-the-art fault localization tools identify suspicious code lines, this information spans across other code entities such as methods and files, which can be sufficient for APR mutations. Thus, to compute the performance of fault localization techniques on a benchmark, we consider different granularities of fault locality at the *file*, *method* and *line* levels similar to the fault locality defined by Lucia et al. [46]:

- **File**: At this level, we consider that the faulty code is accurately localized if an FL tool reports any line from the buggy code file as suspicious.
- **Method**: At this level, we consider that the faulty code is accurately localized if any code line in the buggy method is reported by an FL tool as suspicious.
- **Line**: At this level, we consider that the faulty code is accurately localized if suspicious code lines reported by an FL tool contain any of the buggy code lines.

B. Identification of Correct Fault Locality

Our objective is to identify which reported suspicious code position is correct, following the above three levels of fault locality granularity. In practice, FL tools produce a ranked list of suspicious lines while ground truth data include several code lines as buggy lines as well. At a given granularity level, if the bug is localized (i.e., there is a match between the suspicious code line and the ground truth fault locations), we record the associated position of the correct fault locality within the ranked list of suspicious code locations. Since a bug position could span over several lines, methods, and even over several files, the bug is considered to be correctly localized by an FL tool as long as any reported suspicious code line can match the ground truth bug locations with the corresponding granularity.

Concretely, we first use the following defintion of bug locations. The locations of a bug in a faulty program are defined as a *bug position set*: $BPos$ = $\{bPos_1$, $bPos_2$, ..., $bPos_n\}$, (n >= 1), where $bPos_i$ is a tuple of $(fName$, $Methods$, $Lines)$. For each location, $fName$, $Methods$, and

$Lines$ are a file name, a set of methods, and a list of line numbers, respectively, of a bug location. $Methods$ could be $\emptyset$ if the bug is not located in any method in a program. This kind of bugs can be related to a Type Declaration [47] or Field Declaration [48] in Java code. *Math-12* in the Defects4J dataset is an example, which is fixed by inserting an interface `Serializable` into the type declaration [49].

We then check whether a ranked list of suspicious lines by an FL tool can identify bug locations based on the following definition. Let $SuspL = \{suspL_1, suspL_2, \ldots, suspL_m\}$ be a list of suspicious lines that are reported by an FL tool and ordered by suspiciousness scores. $suspL_i$ is a tuple of ($fName$, $lineNum$, $rIdx$), where $lineNum$ is the line number of the code in a file (i.e., $fName$) that is suspected to be the bug location, and $rIdx$ is the index (i.e., rank) of the line within $SuspL$. If a suspicious line $suspL_i$ ($i \in [1, m]$) matches any bug location ($BPos$) at a given granularity before other suspicious lines, it is considered that the FL tool successfully identifies a bug location at the given granularity. Otherwise, if there is no suspicious line matching a bug location at a given granularity, the fault is considered as non-localizable at this fault locality granularity.

C. Dataset and Automatic Testing Toolset

Our study requires to execute fault localization experiments on a reliable dataset. In this work, we select the Defects4J [34] dataset as it includes test cases for buggy Java programs with the associated developer fixes. This dataset has furthermore been used by all recent state-of-the-art APR systems targeting Java programs. Table III summarizes statistics on the number of bugs and test cases available in the version 1.2.0[1] of Defects4J that we use in this paper.

TABLE III: Defects4J dataset information.

Project	Chart	Closure	Lang	Math	Mockito	Time	Total
# of bugs	26	133	65	106	38	27	395
# of test cases	2,205	7,927	2,245	3,602	1,457	4,130	22,954

# of test cases are excerpted from the Defects4J paper [34] and [50].

Overall, the dataset includes 395 bugs and 22,954 test cases. To automate the execution of these test cases for each bug, we rely on the GZoltar[2] [21] framework for automatic debugging of Java applications. GZoltar executes the test cases and produces coverage matrices providing information on which test cases passed, which failed, which statements were executed when running each test cases, etc. Based on this information, FL techniques can be applied for ranking suspicious code locations which are likely to be the faulty code. For the purpose of our study, we have implemented on-top of GZoltar 41 common ranking metrics [22], [23] for fault localization. Given that Gzoltar has been used by several APR tools in the literature, we expect that our easy-to-configure fault localization toolkit will serve the research community to parameterize fault localization in an APR pipeline.

Our experiments further considered two different versions of GZoltar. The first one is the GZoltar version 0.1.1, which is already used in state-of-the-art APR systems, such as Astor [35], FixMiner [40], ACS [19], ssFix [37] and CapGen [38] among others. On the other hand, the GZoltar version 1.6.0 is used in SimFix [33] since it was recently shown to be effective [42].

D. Implementation of a Baseline APR System

Ideally, we should consider exploring an existing APR system for drawing our reference performance. Unfortunately, we face several challenges: (1) only a few research groups openly release the code or even implementation details of their APR systems; (2) repair steps are often tightly coupled together in implementation, which requires substantial engineering effort for experimental adaptation; (3) proposed approaches generally mix several contributions which are hard to isolate.

We, therefore, propose to implement and share a baseline repair system based on a state-of-the-art publication on Java program repair. We select PAR [7] for its simplicity and the straightforward replication that can be carried out on the basis of details from the relevant research report. We build `kPAR`, which leverages patterns that have been learned from the commonalities among 60,000 human-written patches. Six common patterns from the initial version of PAR has been implemented in `kPAR`. We further record the performance of `kPAR` in repair scenarios involving four different configurations of the fault localization step.

IV. Study Results

We now provide key findings for the related questions that are investigated in this work.

A. Integration of FL Tools in APR Pipelines

To characterize how FL tools are integrated into APR pipelines, we carefully assess evaluation reports in the literature and investigate the source code (whenever it is available) of 14 state-of-the-art APR systems which have been evaluated on the Defects4J benchmark. Table IV enumerates the studied tools along with the information collected. We focus on the testing framework that is used and its version, the FL ranking metric that is considered to compute the suspiciousness scores, the granularity of fault locality that authors focused on, and the extra information that authors use to supplement FL.

Among the 14 APR tools that are investigated, 10 leverage GZoltar as the automated testing toolset in the repair pipeline. Except for SimFix, which uses a recent version of the framework, all others use earlier versions (8 tools use version 0.1.1, while Nopol uses an even older version, i.e., 0.0.1). Thus, unless otherwise stated, the experiments in this work are performed on the widely used version 0.1.1 of GZoltar.

11 out of the 14 APR tools are explicitly known to rely on Ochiai for computing the suspiciousness scores in the fault localization process. This popularity of Ochiai is backed up by empirical evidence on its effectiveness to help localize faults in object-oriented programs as highlighted by several fault localization studies [44], [43], [23], [53]. A recent work by Pearson et al. [42] has even shown that Ochiai outperforms

[1] https://github.com/rjust/defects4j/releases/tag/v1.2.0

[2] http://www.gzoltar.com/.

TABLE IV: Fault Localization (FL) techniques integrated into state-of-the-art APR tools.

	jGP	jKali	jMutRepair	HDRepair	Nopol	ACS	ELIXIR	JAID	ssFix	CapGen	SketchFix	FixMiner	LSRepair	SimFix
FL testing framework	GZoltar	GZoltar	GZoltar	-	GZoltar	GZoltar	-	-	GZoltar	GZoltar	-	GZoltar	GZoltar	GZoltar
Framework version	0.1.1	0.1.1	0.1.1	-	0.0.10	0.1.1	-	-	0.1.1	0.1.1	-	0.1.1	0.1.1	1.6.0
FL ranking metric	Ochiai	Ochiai	Ochiai	-	Ochiai	Ochiai	Ochiai	-	-	Ochiai	Ochiai	Ochiai	Ochiai	Ochiai
Granularity of fault locality	line	line	line	line	line	line	line	line	line	line	line	method	line	line
Supplementary information	∅	∅	∅	Faulty method is known	∅	Predicate switching [51]	∅	∅	Statements in crashed stack trace	∅	∅	∅	∅	Test Case Purification [52]

* The missing specific FL tools used by APR tools are marked with '-'. If an APR tool does not use any supplementary information for fault localization, the corresponding table cell is marked with '∅'.

current state-of-the-art ranking metrics, or at least offers similar performance measures. In the latter part of this study, we replicate their work to ensure that our implementation of the ranking techniques is reliable. It should also be noted that although ELIXIR and SketchFix do not report the test framework that they use, they explicitly mention using Ochiai for fault localization.

With respect to the granularity of fault locality, only LSRepair [41] focuses on the method-level granularity to detect and fix bugs. Other APR systems require information on bugs at the line level to proceed with patch generation. Considering methods as the granularity of fault locations implies that such faults that are located outside methods (e.g., type declaration faults [54]) will not be addressed. However, this granularity may offer a time advantage: when several statements in a single method are reported as suspicious locations, LSRepair, unlike other APR systems, is not required to iteratively try each location for generating patch candidates. Finally, it should be noted that FL tools do not offer the same accuracy in identifying faulty locations at different granularity levels (cf. Section IV-B), making method level granularity appealing for limiting unnecessary trials on fault positive locations.

It is further noteworthy that four APR systems leverage supplementary information to assist the fault localization step and improve accuracy. The impact of this assistance is unfortunately never discussed when comparing performance among state-of-the-art repair approaches. Typically:

- HDRepair [13] assumes that the faulty methods are known: the fault localization step therefore focuses on ranking the lines inside the method, thus leaving out noisy statements that other APR tools are considering. This artificially reduces the probability to produce overfitting patches for HDRepair, and even increases the chance to generate a correct patch before any execution timeout.
- ssFix [37] prioritizes statements from the stack trace of crashed programs that are executed before those statements that are ranked by the FL tool.
- ACS [19] uses predicate switching [51] and refines the suspicious code locations list since the repair is focused on faulty conditional statements.
- SimFix [52] applies a test case purification approach to improve the accuracy of FL step before patch generation.

Although these extra steps, which are taken to supplement FL step, could be justified intuitively, the community needs to clearly investigate their impact, in order to enable fair comparisons among the repair techniques themselves. Indeed, given that APR systems are currently compared with respect to the number of bugs that are correctly fixed, it is important that the research community reflects on what are the key contributions for explaining APR performance: for example, by counting numbers of correct patches, several programs may not be repairable by a given APR system simply because the fault is not accurately localized by the implemented FL step.

> **RQ1**▶*State-of-the-art APR systems in the literature some adaptations to the usual FL process to improve its accuracy. Unfortunately, researchers have eluded so far the contribution of this improvement in the overall repair performance, leading to biased comparisons.*

B. Localizability of Defects4J Bugs

In a recent work, Koyuncu et al. [40] have reported that 136 bugs in total from the Defects4J dataset have already been associated to a plausible patch that was generated by at least one APR system from the literature. Patches for 83 bugs have even been validated as correct patches by researchers. Considering this data that we complement with the performance realized by another recent APR tool, namely LSRepair, we conclude that ∼62% (246/395) of Defects4J's bugs have never seen a plausible patch automatically generated by the state-of-the-art in APR. Although a recent empirical study [55] has suggested that current APR systems cannot repair hard and important bugs, our intuition is that there might be a more practical issue related to the localizability of Defects4J defects:

How many faults in the Defects4J benchmark can actually be localized by current automated fault localization tools?

We consider the most common scenario of fault localization scenario from the APR literature: GZoltar is used for automated test execution, and Ochiai for computing suspiciousness scores. Test execution is performed with the test cases provided in the Defects4J benchmark. Table V provides quantitative details on the localizability of bugs at different levels of fault locality granularity (i.e., file, method and line). Experiments are performed with two distinct versions of GZoltar.

In this experiment, we consider a bug to be localized as long as the faulty code is listed among the suspicious statements reported by this fault localization tools. Considering the most common configuration in the literature (GZoltar version 0.1.1 and "Line" granularity level), up to 132 (= 395 - 263) bugs in Defects4J are not localized. The number bugs that are not localized decreases to 74 (= 395 - 321) when the coverage matrices are produced with GZoltar version 1.6.0. This result

TABLE V: Number of Bugs localized* with Ochiai/GZoltar.

Project	# Bugs	File		Method		Line	
		GZ_1	**GZ_2**	**GZ_1**	**GZ_2**	**GZ_1**	**GZ_2**
Chart	26	25	25	22	24	22	24
Closure	133	113	128	78	96	78	95
Lang	65	54	64	32	59	29	57
Math	106	101	105	92	100	91	100
Mockito	38	25	26	22	24	21	23
Time	27	26	26	22	22	22	22
Total	395	344	374	268	325	263	321

*A bug is counted as localized as long any of the faulty locations appear in the ranked list of suspicious locations reported by the FL tool. GZ_1 and GZ_2 indicate GZoltar 0.1.1 and 1.6.0, respectively. The same abbreviations are used for GZoltar versions in the following tables. The column GZ_1 of "Line" is highlighted since it is the most common configuration in APR systems.

suggests that with GZoltar version 1.6.0, APR systems have an opportunity attempt the fix of 58 more bugs.

> **RQ2**►*One third of bugs in the Defects4J dataset cannot be localized by the commonly used automated fault localization tool. Nevertheless, the recent version of GZoltar provide coverage information that helps localize more than 50 bugs, which may have never been considered in validation trials of early APR systems.*

Besides Ochiai, we have attempted to localize bugs in the Defects4J benchmark by using six other ranking metrics to compute suspiciousness scores. Table VI presents the number of bugs localized by the different ranking metrics. We consider the cases where the actual fault location is reported at the Top-1 position of the suspicious code locations, and among the Top-10 positions. Results for Top-50, Top-100, Top-200 and all localized are also made available in the replication package. The results show that fault localization performance is consistent among the different ranking metrics.

TABLE VI: Number of Bugs localized at Top-1 and Top-10.

Ranking Metric	**GZ^1**			**GZ^2**		
	File	**Method**	**Line**	**File**	**Method**	**Line**
Top-1 Position						
Tarantula	171	101	45	169	106	35
Ochiai	173	102	45	172	111	38
DStar2	173	102	45	175	114	40
Barinel	171	101	45	169	107	36
Opt2	175	97	39	179	115	39
Muse	170	98	40	178	118	41
Jaccard	173	102	45	171	112	39
Top-10 Position						
Tarantula	240	180	135	242	189	144
Ochiai	244	184	140	242	191	145
DStar2	245	184	139	242	190	142
Barinel	240	180	135	242	190	145
Opt2	237	168	128	239	184	135
Muse	234	169	129	239	186	140
Jaccard	245	184	139	241	188	142

Only 45 bugs can be accurately localized with Ochiai at the first suspicious line location. 140 and 214 bugs can be localized at Top-10 and Top-100 positions. Actually, many APR systems only focus on generating patches iteratively based on a part of the list of suspicious code locations. For example, for SketchFix [39], authors explicitly declare to consider only the top-50 most suspicious statements in the ranked list, while in ELIXIR [36], up to the top-200 suspicious locations are considered.

C. Impact of Effective Ranking in Fault Localization

Automated fault localization produces a ranked list of suspicious code locations that APR tools must iteratively consider for patch generation. To assess to what extent effective ranking (i.e., placing the actually faulty code locations at the top of the list), we propose to investigate the correlation between the rank of bug localization in the suspicious lists and the ability of state-of-the-art systems to be able to repair it.

Table VII summarizes the list of all bugs, from the Defects4J benchmark, for which a plausible patch has been generated by one of the 14 state-of-the-art APR systems considered in this study. For each bug, we indicate the rank of the bug location within the ranked list of suspicious locations provided by the fault localization for different localization granularities. Experiments are done using the Ochiai ranking metric, but with two versions of GZoltar for computing the test coverage matrices. The raw data, including for other ranking metrics, are available in our replication package.

We propose to compute the distributions of positions across subsets of bugs for checking correlations between the localization ranking positions and the ability of APR systems to fix the bugs. Thus, we normalize bug localization positions by computing reciprocal positions based on the following formula:

$$Reciprocal_{pos}(bug_{pos}) = \begin{cases} 0, & \text{if } bug_{pos} = 0; \\ \frac{1.0}{bug_{pos}}, & otherwise. \end{cases} \quad (2)$$

where bug_{pos} refers to the position of the actual bug location[3] in the ranked list of suspicious locations reported by the FL step. If the bug location can be found in the higher position of the ranked list, the value of $Reciprocal_{pos}$ is closer to 1. Similarly, the value of $Reciprocal_{pos}$ trends to 0 when the bug location is at lower positions in the list of suspicious locations. This value is set to 0 when the bug cannot be localized by the FL tool (i.e., $bug_{pos} = 0$). In addition, for the purpose of our experiments, we consider three sub-classes of bugs:

- *correctly fixed bugs*: these are bugs for which a correct patch has been provided by at least one APR tool.
- *overfitting-fixed bugs*: these are bugs for which one or more plausible patch has been generated, although none has been found to be correct.
- *unfixed bugs*: these are bugs for which no plausible patch has ever been generated by any APR system. Due to space limitation, localization data for these bugs are only available in the replication package.

Figure 2[4] shows the distribution of reciprocal positions for the three classes of bugs at the file, method, line granularity of fault locality. It clearly appears that correctly-fixed bugs are more accurately localized than others: i.e., their location precisions are higher in the ranked list of suspicious locations by FL tools. On the other hand, unfixed bugs tend to be those

[3] If several lines are concerned by the bugs, we consider the first time any of these lines appear as the bug position (cf. Section II).

[4] The bug positions before being reciprocated shown in the figure are localized by GZoltar 0.1.1 with Ochiai.

TABLE VII: Localization positions (i.e., rank within the suspicious list) for Defects4J bugs which have been fixed (plausibly and correctly) by APR systems.

	APR tools														GZ^1 & Ochiai			GZ^2 & Ochiai		
Bug ID	jGenProg	jKali	jMutRepair	HDRepair	Nopol	ACS	ELIXIR	JAID	ssFix	CapGen	SketchFix	FixMiner	LSRepair	SimFix	File	Method	Line	File	Method	Line
Chart-1	○	○	●	○			●	(●)	●	●	●	●	●	●	1	1	28	24	24	28
Chart-3	○				○		○							●	1	1	7	1	1	4
Chart-4												●	●		1	1	49	2	2	173
Chart-5	○	○			●				○						1	1	7	12	66	72
Chart-6													○		2	49	49	24	222	224
Chart-7	○		○						○					●	1	2	28	1	2	75
Chart-8				○			●			●	●				0	0	0	1	1	1
Chart-9							●	(●)	○		●				1	1	3	1	2	14
Chart-10													○		1	1	1	1	3	3
Chart-11							●			●	●	●	●		1	1	15	1	24	28
Chart-12												○		○	1	0	0	801	1092	1093
Chart-13	○	○			○		○		○		○	○			1	1	17	5	35	51
Chart-14						●								○	1	1	1	38	996	998
Chart-15	○	○													1	26	26	2143	8444	8445
Chart-17							○						○		1	2	2	11	12	12
Chart-18													○	○	1	1	6	1	1	3
Chart-19						●									1	1	5	37	833	833
Chart-20									●		●			●	4	0	0	62	62	62
Chart-21					○										2	2	2	1	34	39
Chart-22														○	1	1	1	1	53	58
Chart-24								●	●	●	●	●			1	1	2	1	1	3
Chart-25	○	○	○		○							○			1	30	47	1325	3668	3913
Chart-26		○	○		○			●			○	●	○		132	132	132	241	14795	15053
Closure-5								○							8	0	0	561	0	0
Closure-7									○						7	0	0	28	0	0
Closure-10				○								●			3	56	120	3	67	141
Closure-12									○						154	368	368	393	1085	1085
Closure-14				○					●		●			●	1	2	3	2	3	3
Closure-18								●							90	1495	1527	93	2320	2377
Closure-31								(●)							215	1026	1043	214	1756	1802
Closure-33								●							2	2	289	2	2	318
Closure-38												●			1	1	34	1	1	49
Closure-40								●							9	0	0	104	0	0
Closure-42									○						4	0	0	15	0	0
Closure-51				○											25	25	33	41	41	50
Closure-57														●	1	2	3	1	2	7
Closure-62				○				(●)			●	●		●	1	1	1	1	1	4
Closure-63								(●)				●		●	1	1	1	1	1	4
Closure-68									○						2	2	2	2	2	4
Closure-70				○				●			○				143	0	0	264	0	0
Closure-73				○				●			○	●		●	1	7	10	1	1	16
Closure-79														○	0	0	0	1	37	37
Closure-106														○	0	0	0	3	4	4
Closure-109									○						1	9	9	1	4	4
Closure-111									○						1	0	0	7	0	0
Closure-115									●					●	1	1	1	8	8	8
Closure-122									○						1	1	2	1	2	2
Closure-125								○	○						4	142	145	5	166	170
Closure-126				○				(●)	○		●				1	1	1	6	6	6
Lang-2									○				○		127	127	128	1	1	17
Lang-6				●			●		●	●	●				0	0	0	54	73	74
Lang-7						●									373	0	0	1	1	25
Lang-10				○										○	114	0	0	1	64	64
Lang-16														●	322	0	0	1	1	27
Lang-21									●				●		1	0	0	1	1	2
Lang-24						●	●	○					●		259	618	0	1	14	64
Lang-26							●			●					21	0	0	1	112	112
Lang-27			○						○					●	262	269	0	1	1	56
Lang-29													●		59	59	0	1	1	0
Lang-33							●	●	●					●	14	0	0	1	1	7
Lang-35						●									53	0	0	1	1	2
Lang-38							●	(●)							104	0	0	1	3	3
Lang-39					○	○	○	○	○					●	203	0	0	1	2	27
Lang-40													○		1	1	1	1	1	2
Lang-41													○	●	1	5	7	1	5	6
Lang-43				○			●		●	●		○		●	1	1	1	1	26	29
Lang-44					●		○		○					○	1	5	20	1	1	3
Lang-45								(●)						○	1	1	16	1	1	5
Lang-46					○								●		1	1	3	1	1	1
Lang-48													●		1	1	2	1	1	2
Lang-50														●	1	9	15	1	8	8
Lang-51				●	○		○	(●)	○		○		●		1	1	0	1	1	0
Lang-52													●		1	1	13	1	3	25
Lang-53					○										1	1	32	1	1	16
Lang-54													●		1	1	2	1	1	4
Lang-55					●			(●)			●		○		1	7	9	1	6	7
Lang-57				○			●			●		●			1	1	1	1	1	1
Lang-58					●		○		○					●	1	2	8	1	1	20
Lang-59				○			●		●	●	●	●			1	1	1	1	1	6
Lang-60													○	●	1	1	3	1	1	2
Lang-61								○							1	14	19	1	14	21
Lang-62													○		1	1	7	1	1	7
Lang-63									○					○	1	1	1	1	1	1
Math-1														○	20	0	0	12	14	14
Math-2	○	○	○				○		○						1	11	11	44	44	44
Math-3						●			○						1	1	16	1	1	3
Math-4						●									1	1	1	1	3	6
Math-5	●			●		●	●	●		●	●			●	1	2	2	1	1	1
Math-6									○					○	1	2	205	1	2	177
Math-8	○	○							○					○	1	1	3	1	3	4
Math-10												●			1	1	1	5	5	17
Math-11													○		1	9	9	27	27	29
Math-16													○		1	1	5	1	1	5
Math-20							○		○					○	1	0	0	1	0	0
Math-22				●								●			1	1	1	1	1	1
Math-25						●									0	0	0	0	0	0
Math-28	○	○	○			○			○				○	○	6	6	6	13	13	13
Math-30							●		●	●		●			1	4	9	142	161	161
Math-32		○			○		○	(●)							1	3	3	5	5	6
Math-33					○		●		●	●	●	●		●	2	4	31	2	6	44
Math-34				○			●					●			1	1	1	1	3	3
Math-35						●								●	0	0	0	1	3	5
Math-40	○	○	○		○										1	23	24	24	33	34
Math-41									●					●	1	2	6	1	37	49
Math-42					○										1	23	26	3	57	66
Math-49	○	○			○										3	4	7	5	5	7
Math-50	●	●	○	●	●		●	(●)	●		●			●	1	1	1	1	1	1
Math-53	●			●				(●)	●	●				●	1	1	1	1	1	2
Math-57			○		○		●		●	●		●		●	1	1	14	1	4	4
Math-58			○		○		●		○	●		●			6	6	6	223	223	223
Math-59							●		●	●	●			●	1	1	1	1	2	2
Math-60									○						21	21	21	283	283	284
Math-61						●									0	0	0	1	1	1
Math-63							○		○	●			●	●	1	8	8	1	1	1
Math-65									○	●					1	8	9	12	12	15
Math-69					○										1	1	2	1	48	57
Math-70	●			○			●		●	●	●	●	●	●	1	1	1	1	1	1
Math-71	○				○				○					●	1	1	1	1	1	1
Math-72														○	1	3	4	1	1	1
Math-73	●				○	○	○				○			○	1	1	1	1	1	1
Math-75							●			●		●	●	●	1	2	2	1	1	1
Math-78	○	○			○				○						17	21	32	67	67	109
Math-79									●	○		●	●	●	23	23	25	29	29	29
Math-80	○	○			○		○	(●)	●	○			○	○	1	11	18	1	14	14
Math-81	○	○	○		○	○			○	○		○		○	1	1	6	1	1	10
Math-82	○	○	●	○	○	●	●	(●)		○	●	●		○	2	53	60	1	76	84
Math-84	○	○	○						○			○			1	13	30	5	18	134
Math-85	○	○	●		○	●	●	(●)	○	●	●	●		○	1	1	36	11	11	90
Math-87					○										1	99	100	2	109	111
Math-88			○		○									○	1	1	1	1	1	1
Math-89						●							●		1	1	1	1	1	1
Math-90						●									1	1	4	1	1	3
Math-91													●		1	1	2	1	1	1
Math-93						●							○		1	1	2	1	1	2
Math-94													●		1	1	21	1	1	21
Math-95	○	○							○				○		2	2	3	8	11	12
Math-97					○	○									1	1	1	1	1	1
Math-98														●	1	1	6	1	1	4
Math-99						●							○		1	1	1	1	1	4
Math-104					○		○								1	0	0	1	0	0
Math-105					○			○							1	1	1	1	25	25
Mockito-13													●		30	30	70	74	74	135
Time-4	○	○					●		○		○				36	36	208	6	6	31
Time-7														●	48	48	51	10	10	14
Time-11	○	○	○		○		○		○						4	0	0	51	0	0
Time-14									○						4	4	7	2	2	3
Time-15						●	●								1	1	115	1	1	2
Time-17									○						5	5	5	5	5	5
Time-19				○								●			5	449	449	104	620	620

∗ ● indicates that the bug is correctly fixed and ○ indicates that the generated patch is plausible but not correct. (●) indicates that a correct patch is generated, but is not the first plausible patch to be generated". "0" means that the bug cannot be localized by the corresponding FL tool with the corresponding ranking metric in the corresponding granularity.

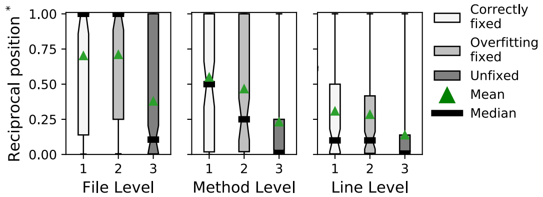


Fig. 2: Distribution of reciprocal positions of actual bug locations among the ranked list of suspicious locations.

that are poorly localized: even at the file level, FL tool show low performance in localizing such bugs.

> **RQ2▶***APR tools are prone to correctly fix the subset of Defects4J bugs that can be accurately localized.*

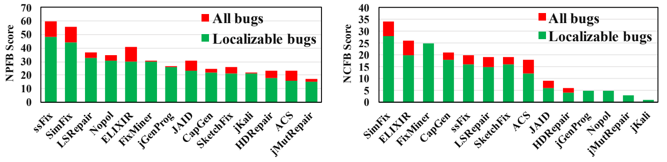
We further observe from the data in Table VII that a few APR systems report patches for some bugs even though they cannot be localized (at the line level) with the configuration of Ochiai/GZoltar 0.1.1. There are various justifications to this phenomenon:

- **Improved version of the fault localization step** - *Chart-20* cannot be localized with GZoltar 0.1.1 and Ochiai, but is reported to be fixed by tools such as SimFix and ssFix. Our investigations show that SimFix has used a recent version of GZoltar (1.6.0), which is capable of localizing *Chart-20* among other bugs that were not localizable. ssFix on the other hand indeed uses GZoltar 0.1.1 but do not consider only the results of the FL tool: statement in the stack trace of crashed programs are also considered as potential fault locations.
- **Targeted localization** - HDRepair can fix *Lang-6*, which is not localized with Ochiai/GZoltar 0.1.1, because this APR system assumes that the faulty method is known, and thus directly ranks the restricted set of statements in this method.
- **Coarse-grained repair** - LSRepair can fix four bugs which cannot be localized at the line granularity. This is due to the fact that LSRepair requires only fault localization at the method level, which is not a bias per se.
- **Non-explicit fault localization process** - SketchFix, JAID, and ELIXIR correctly fix some bugs that are not localized under the proposed configuration. Unfortunately, besides the lack of details in their associated research reports, the source code of these tools was not made available for further investigation. *Chart-8* is another example that is not localizable by using Ochiai/GZoltar 0.1.1. This specific un-localizability problem was recently raised by Yuan and Banzhaf [56] as well as Martinez et al. [45]. Nevertheless, CapGen, ELIXIR and SketchFix are reported to have fixed this bug.

> **RQ3▶***APR systems do not fully disclose their fault localization tuning parameters, thus preventing reliable replication and comparisons.*

Given the bias that can be introduced by unlocalizable bugs being fixed by specific tweaking, which are not clearly outlined by the authors, we propose to count the numbers of bugs that are fixed by APR systems among those bugs that are known to be localizable. Table IX thus represents an updated version of Table II where performance can be compared on the same basis. To illustrate the differences between the two comparison tables, we compute three scores: (1) **NPFB**: number of plausibly-fixed bugs, (2) **NCFB**: number of correctly-fixed bugs, and (3) **P^{3}C**: probability of plausible patch correctness.

Figures 3a and 3b illustrate the differences in respectively NPFB and NCFB scores when considering all bugs vs only localizable bugs. We note that all tools may produce some plausible patches that are plausible even for non-localizable bugs. This finding suggests that the test cases in Defects4J are insufficient since it is possible for APR systems to change non-faulty code locations and still produce patches that make the faulty program pass all test cases. On the other hand, five APR systems cannot produce any correct patches for bugs that are not localizable. ACS, ELIXIX and SimFix can correctly fix bugs that are not localized with GZoltar 0.1.1, suggesting extra impact with an improved version of the fault localization step. On the other hand, LSRepair can fix bugs that are not localized at the line level because method level fault localization is sufficient for its execution.



(a) # of plausibly-fixed bugs. **(b)** # of correctly-fixed bugs.

Fig. 3: Number of fixed bugs among all bugs vs. localizable bugs.

Finally, Table VIII establishes the re-ranking of APR systems in terms of the **P^{3}C** scores when focusing on localizable bugs. When focusing on localizable bugs, state-of-the-art APR systems can correctly overall fix fewer bugs than reported in the literature.

TABLE VIII: Adjusted Probability of Plausible Patch Correctness.

All			Localizable	
P^{3}C	Rank	Tool	P^{3}C	Rank
84.0	1	CapGen	81.8	↓ 2
80.6	2	FixMiner	83.3	↑ 1
78.3	3	ACS	75.0	↓ 4
73.1	4	SketchFix	76.2	↑ 3
63.4	5	ELIXIR	66.7	5
60.7	6	SimFix	63.6	6
51.4	7	LSRepair	45.5	7
33.3	8	ssFix	33.3	8
29.0	9	JAID	26.1	9
26.1	10	HDRepair	22.2	10
18.5	11	jGenProg	19.2	↓ 12
17.6	12	jMutRepair	20.0	↑ 11
14.3	13	Nopol	16.1	13
4.5	14	jKali	4.8	14

D. *Evaluating `kPAR` with Specific FL Configurations*

`kPAR` is an open-source APR system that we have built to provide a baseline for comparisons of different FL configurations. We evaluate its performance against the Defects4J benchmark with the following four different configurations of the fault localization step:

TABLE IX: Number of localizable bugs (with GZoltar 0.1.1 and Ochiai) fixed by different APR tools.

Proj.	**jGenProg**	**jKali**	**jMutRepair**	**HDRepair**	**Nopol**	**ACS**	**ELIXIR**	**JAID**	**ssFix**	**CapGen**	**SketchFix**	**FixMiner**	**LSRepair**	**SimFix**
Chart	0/7	0/6	1/4	0/1	1/6	2/2	3/6	2/4	2/6	3/3	4/6	5/7	3/8	3/6
Closure	0/0	0/0	0/0	0/6	0/0	0/0	0/0	3/8	2/8	0/0	3/4	5/5	0/0	6/6
Lang	0/0	0/0	0/0	0/3	3/5	0/0	3/5	0/3	2/6	3/3	2/2	2/3	4/10	5/8
Math	5/18	1/14	2/11	4/7	1/20	9/13	12/17	1/8	10/25	12/16	7/8	12/14	7/14	13/23
Mockito	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0
Time	0/1	0/1	0/0	0/1	0/0	1/1	2/2	0/0	0/3	0/0	0/1	1/1	0/0	1/1
Total	5/26	1/21	3/15	4/18	5/31	12/16	20/30	6/23	16/48	18/22	16/21	25/30	15/33	28/44
Total* (all bugs)	5/27	1/22	3/17	6/23	5/35	18/23	26/41	9/31	20/60	21/25	19/26	25/31	19/37	34/56
P(%)	19.2	4.8	20.0	22.2	16.1	75.0	66.7	26.1	33.3	81.8	76.2	83.3	45.5	63.6
P(%)* (all bugs)	18.52	4.55	17.65	26.09	14.29	78.26	63.41	29.03	33.33	84.00	73.08	80.65	51.35	60.71

*Greyed-out rows are copied from Table II (i.e., numbers reported in the literature) to ease comparison with the numbers of localizable bugs that are fixed.

1) **Normal_FL** gives a ranked list of suspicious code locations identical as reported by a given FL tool.
2) **File_Assumption** assumes that the faulty code files are known. Suspicious code locations from Normal_FL are then filtered accordingly. In other words, locations in the known buggy files are selected and locations in other files are ignored.
3) **Method_Assumption** assumes that the faulty methods are known (the same assumption with [13]). Only locations in the known methods are selected and locations in other methods are ignored.
4) **Line_Assumption** assumes that the faulty code lines are known. No fault localization is then used.

These configurations have an order with respect to a potential size of the search space. Conceptually, the relationships between them hold $P(|Normal\_FL|) \leq P(|File\_Assumption|) \leq P(|Method\_Assumption|) \leq P(|Line\_Assumption|)$, if we consider each configuration as producing a set of suspicious locations, where $P(|*|)$ is the probability that the relevant fault locations are included in the suspicious list.

To facilitate comparison with existing repair systems, we leverage the standard GZoltar 0.1.1 and Ochiai in the following experiments. For each bug, we apply kPAR at most three hours (wall-clock time); we assume that it fails to fix a given bug if it takes more than three hours. We set this value according to the experimental setup of Astor [35]. Table X summarizes the number of bugs fixed by kPAR with the different FL configurations.

As shown in Table X, kPAR can fix its maximum number of bugs when the accurate fault locations are provided (i.e., with *Line_Assumption*). With this assumption, kPAR can correctly fix 36 bugs in Defects4J, a record performance in the literature (not accounting for the bias in the fault localization step).

TABLE X: # of Bugs fixed by kPAR.

FL Configuration	**Chart (C)**	**Closure (Cl)**	**Lang (L)**	**Math (M)**	**Mockito (Moc)**	**Time (T)**	**Total**
Normal_FL	3/10	5/9	1/8	7/18	1/2	1/2	18/49
File_Assumption	4/7	6/13	1/8	7/15	2/2	2/3	22/48
Method_Assumption	4/6	7/16	1/7	7/15	2/2	2/3	23/49
Line_Assumption	7/8	11/16	4/9	9/16	2/2	3/4	36/55

Figure 4 further details which bugs are fixed in the different configurations. First, we note that all bugs fixed with a given localization configuration are also fixed by any of the relatively more accurate fault localization configurations. Thus, with the *File_Assumption* configuration, kPAR can fix not only all bugs that were already fixed with the *Normal_FL* configuration but also can now fix four more bugs. By examining the case of those four bugs, we figure out that, in the case of two bugs (i.e., *Cl-4* and *T-19*), the faulty locations were ranked very low in *Normal_FL*, leading to an execution stop due to timeout. For the remaining two bugs (i.e., *C-26* and *Moc-29*), however, in *Normal_FL*, kPAR is led to consider first some irrelevant suspicious statements that made kPAR to generate plausible patches that are not correct. Given that the repair process stops when a plausible patch is produced, there is no opportunity with *Normal_FL* to try all suspicious statements.

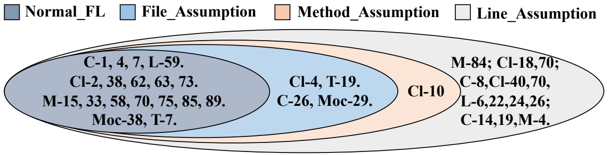


Fig. 4: Bugs correctly fixed by kPAR with four configurations.

When filtering the set of suspicious locations with *Method_Assumption*, kPAR can fix one more bug (i.e., *Cl-10*), which could not be fixed by other two less confined FL configurations (i.e., *Normal_FL* and *File_Assumption*) before the time-out. Finally, when assuming that the fault locations are known (i.e., *Line_Assumption*), kPAR can further fix 13 bugs. These could not be fixed in other three less confined configurations. Among the 13 bugs, seven bugs (i.e., *C-8*, *Cl-40*, *Cl-70*, *L-6*, *L-22*, *L-24*, and *T-26*) are not even localizable using Ochiai/GZoltar 0.1.1; two bugs (i.e., *Cl-18* and *Cl-70*) are not fixed due to execution timeout; one bug (i.e., *M-82*) is not fixed in other three configurations since the proposed plausible patches are incorrect; three bugs (i.e., *C-14*, *C-19* and *M-4*) are partially fixed in the other three FL configurations since they have several faulty code fragments.

> **RQ3►***Accuracy of fault localization has a direct and substantial impact on the performance of APR repair pipelines.*

We examine the bug *Chart-14* from the Defects4J dataset, which involves four fault code locations [57]. If we regard those as four sub-bugs, each one can be correctly detected and fixed by kPAR using the *Normal_FL* configuration. However, if the exact faulty statements are unknown, kPAR (as current APR tools) iteratively mutates suspicious statements one by one in the ranked list. Even if any one of them is correctly fixed, there are still three failed tests, meaning that the generated patch (even if was a correct patch) will not even be considered as a plausible patch.

Considering a patch that partially passes some previously-failing test cases (without introducing new failing test cases) may nevertheless be harmful as it can prevent the generation of a fully correct patch. For example, *Chart-4* is a single-location bug that makes 22 test cases fail [58]. Before generating the correct patch, `kPAR` had generated patches that made the program pass subsets of the test cases.

Other bugs, such as *Math-72*, on the other hand include multiple faulty locations that fail on the same test case. Although `kPAR` could generate correct patches for each faulty location, the fix process of `kPAR` prevents a full fix of this bug. If the test suite can be automatically augmented with differentiating test cases for each fault location, an APR system would be more successful as suggested in [30].

> **RQ3**►*APR researchers must investigate the trade-off between fixing multi-locations bugs versus bugs failing multiple test cases.*

V. Discussion

Our study draws a number of conclusions that we reformulate into guidelines for assessing APR systems. We further enumerate the associated threats to validity before discussing the related work.

A. APR Assessment Guidelines

- **Full disclosure of FL parameters.** Given that many APR systems do not release their source code, it is important that the experimental reports clearly indicate the protocol used for fault localization. Preferably, authors should strive to assess their performance under a standard and replicable configuration of fault localization.
- **Qualification of APR performance.** To ensure that novel approaches to APR are indeed improving over the state-of-the-art, authors must qualify the performance gain brought by the different ingredients of their approaches.
- **Patch generation step vs Repair pipeline.** There are two distinct directions of repair benchmarking that APR researchers should consider. In the first, a novel contribution to the patch generation problem must be assessed directly by assuming a perfect fault localization. In the second, for ensuring realistic assessment w.r.t. industry adoption, the full pipeline should be tested with no assumptions on fault localization accuracy.
- **Sensitivity of search space.** Given that fault localization produces a ranked list of suspicious locations, it is essential to characterize whether exact locations are strictly necessary for the APR approach to generate the correct patches. For example, an APR system may not focus only on a suspected location but on the context around this location. APR approaches may also use heuristics to curate the FL results.

B. Threats to Validity

A threat to external validity of our study is that we focus on the localizability of bugs in the Defects4J dataset, which target Java code and may not include sufficient test cases. This threat is however limited given that we investigate performance differences. Threats to internal validation include the use of a single automatic testing framework, namely GZoltar (Not all APR systems in the literature use it to localize faults.), and the selection of the 14 state-of-the-start APR systems. These threats are mitigated by the fact that we ensured that these choices are common among the APR literature.

C. Related Work

Various studies in the literature have explored the effectiveness of fault localization [59], [25], [60], [61], [46], [24], [62], [42]. We now discuss the few related studies that attempt to investigate fault localization in relationship with APR.

Qi et al. [63] have evaluated the effectiveness of FL tools by using APR performance as a proxy. Their study proposed the NCP score [63] as the effectiveness metric. The results show that a specific FL ranking metric (Jaccard [64]) outperforms other metrics. Our study, however, reveals that the common technique used in APR is still Ochiai. Yang et al. [65] studied the usage of FL techniques in APR systems by investigating two different algorithms of how to interpret the results of FL techniques: (1) the rank-first algorithm based on suspiciousness rankings of statements, and (2) the suspiciousness-first algorithm based on suspiciousness scores of statements. They ran Nopol [18] to compare NCP scores, repair time, and patch diversity of the two algorithms. The study concludes that the suspiciousness-first algorithm is more effective for APR systems. The above two studies, however, do not consider whether the patches generated by APR tools are correct or plausible while our study examines how FL techniques affect the quality of patches generated by APR systems.

The literature also includes work on the impact of the fault space, although it does not clarify how FL tools affect the performance of APR systems. Wen et al. [66] investigated the influence of the fault space on the success of finding correct patches by the APR tool. The fault space is defined as a ranked list of suspicious entities in a program. They examined both plausible and correct patches. However, their work is limited to evaluating a single APR tool, GenProg [6] and a single FL technique, Ochiai [25] while our study evaluates and compares 14 different APR systems. Our study further considers the exact location of faults and its correlation with the possibility of generating plausible patches. Finally, our study targets unveiling biases among APR systems.

To the best of our knowledge, our work is the first time to systematically study to what extent FL techniques impact the performance of automated program repair pipeline.

VI. Conclusion

The momentum of research in automated program repair is a decisive opportunity for the software engineering research community. Every couple of months, a new APR system is proposed in a race to fix more bugs automatically. Unfortunately, validation of these systems often have only the dataset in common: important parameters such as the fault localization settings are eluded, leading to biased comparisons among the state-of-the-art. Our investigations into these biases call for new guidelines for assessing and reporting on the performance of APR systems. In particular, our replication package includes a full dissection of the Defects4J benchmark in terms of fault localization, a light-weight and tuneable fault localization toolkit, as well as a baseline Java APR system to encourage fair and reproducible experiments.

REFERENCES

[1] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix: Fixing common c language errors by deep learning." in *AAAI*, 2017, pp. 1345–1351.

[2] S. Bhatia and R. Singh, "Automated correction for syntax errors in programming assignments using recurrent neural networks," *arXiv preprint arXiv:1603.06129*, 2016.

[3] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 298–309.

[4] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: program repair via semantic analysis," in *Proceedings of the 35th International Conference on Software Engineering*. San Francisco, CA, USA: IEEE, 2013, pp. 772–781.

[5] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering, May 16-24,*. Vancouver, Canada: IEEE, 2009, pp. 364–374.

[6] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[7] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 35th International Conference on Software Engineering*. San Francisco, CA, USA: IEEE, 2013, pp. 802–811.

[8] Z. Coker and M. Hafiz, "Program transformations to fix c integers," in *Proceedings of the International Conference on Software Engineering*. San Francisco, CA, USA: IEEE, 2013, pp. 792–801.

[9] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Lincoln, NE, USA: IEEE, 2015, pp. 295–306.

[10] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. Florence, Italy: IEEE, 2015, pp. 448–458.

[11] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Bergamo, Italy: ACM, 2015, pp. 166–178.

[12] X.-B. D. Le, Q. L. Le, D. Lo, and C. Le Goues, "Enhancing automated program repair with deductive verification," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. Raleigh, NC, USA: IEEE, 2016, pp. 428–432.

[13] X. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER*, vol. 1. Suita, Osaka, Japan: IEEE, 2016, pp. 213–224.

[14] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. St. Petersburg, FL, USA: ACM, 2016, pp. 298–312.

[15] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. Urbana, IL, USA: IEEE, 2017, pp. 637–647.

[16] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. Paderborn, Germany: ACM, 2017, pp. 593–604.

[17] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. Paderborn, Germany: ACM, 2017, pp. 727–739.

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

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

[20] K. Liu, A. Koyuncu, D. Kim, and 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.

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

[22] Z. Zhang, W. K. Chan, T. Tse, Y.-T. Yu, and P. Hu, "Non-parametric statistical fault localization," *Journal of Systems and Software*, vol. 84, no. 6, pp. 885–905, 2011.

[23] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 191–200.

[24] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[25] R. Abreu, A. J. Van Gemund, and P. Zoeteweij, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques: MUTATION, 2007. TAICPART-MUTATION 2007*. IEEE, 2007, pp. 89–98.

[26] 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.

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

[28] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 789–799.

[29] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 24–36.

[30] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 831–841.

[31] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 117–128.

[32] X. B. D. Le, F. Thung, D. Lo, and C. Le Goues, "Overfitting in semantics-based automated program repair," *Empirical Software Engineering*, pp. 1–27, 2018.

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

[34] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 437–440.

[35] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 441–444.

[36] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 2017, pp. 648–659.

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

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

[39] J. Hua, M. Zhang, K. Wang, and 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.

[40] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *arXiv preprint arXiv:1810.01791*, 2018.

[41] K. Liu, K. Anil, K. Kim, D. Kim, and T. F. Bissyandé, "LSRepair: Live search of fix ingredients for automated program repair," in *Proceedings of the 25th Asia-Pacific Software Engineering Conference*, Nara, Japan, 2018.

[42] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 609–620.

[43] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, p. 31, 2013.

[44] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 314–324.

[45] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[46] Lucia, F. Thung, D. Lo, and L. Jiang, "Are faults localizable?" in *Proceedings of the 2012 9th IEEE Working Conference on Mining Software Repositories*, 2012, pp. 74–77.

[47] Eclipse, "TypeDeclaration," http://help.eclipse.org/neon/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/TypeDeclaration.html, Last Access: Oct. 2018.

[48] ——, "FieldDeclaration," http://help.eclipse.org/neon/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/FieldDeclaration.html, Last Access: Oct. 2018.

[49] Defects4J, "Math-12," http://program-repair.org/defects4j-dissection/#!/bug/Math/12, Last Access: Oct. 2018.

[50] R. Just, C. Parnin, I. Drosos, and 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.

[51] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 272–281.

[52] J. Xuan and 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.

[53] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[54] K. Liu, D. Kim, L. Li, K. Anil, T. F. Bissyandé, and Y. L. Traon, "A closer look at real-world patches," in *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution*, Madrid, Spain, 2018, pp. 304–315.

[55] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun, "Do automated program repair techniques repair hard and important bugs?" *Empirical Software Engineering*, vol. 23, no. 5, pp. 2901–2947, 2018.

[56] Y. Yuan and W. Banzhaf, "ARJA: Automated repair of java programs via multi-objective genetic programming," *arXiv preprint arXiv:1712.07804*, 2017.

[57] Defects4J, "Chart-14," http://program-repair.org/defects4j-dissection/#!/bug/Chart/14, Last Access: Oct. 2018.

[58] ——, "Chart-4," http://program-repair.org/defects4j-dissection/#!/bug/Chart/4, Last Access: Oct. 2018.

[59] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005, pp. 273–282.

[60] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[61] T. Janssen, R. Abreu, and A. J. van Gemund, "Zoltar: A toolset for automatic fault localization," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 662–664.

[62] A. Perez, R. Abreu, and A. van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 654–664.

[63] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 191–201.

[64] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic Internet services," in *Proceedings International Conference on Dependable Systems and Networks*, Jun. 2002, pp. 595–604.

[65] D. Yang, Y. Qi, and X. Mao, "An empirical study on the usage of fault localization in automated program repair," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 504–508.

[66] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "An empirical analysis of the influence of fault space on search-based automated program repair," *arXiv preprint arXiv:1707.05172*, 2017.