The Patch Overfitting Problem in Automated Program Repair: Practical Magnitude and a Baseline for Realistic Benchmarking

Justyna Petke
University College London
London, UK
j.petke@ucl.ac.uk

Matias Martinez
Universitat Politècnica de Catalunya-BarcelonaTech
Barcelona, Spain
matias.martinez@upc.edu

Maria Kechagia
University College London
London, UK
m.kechagia@ucl.ac.uk

Aldeida Aleti
Monash University
Melbourne, Australia
aldeida.aleti@monash.edu

Federica Sarro
University College London
London, UK
f.sarro@ucl.ac.uk

ABSTRACT

Automated program repair techniques aim to generate patches for software bugs, mainly relying on testing to check their validity. The generation of a large number of such *plausible* yet incorrect patches is widely believed to hinder wider application of APR in practice, which has motivated research in automated patch assessment. We reflect on the validity of this motivation and carry out an empirical study to analyse the extent to which 10 APR tools suffer from the *overfitting problem* in practice. We observe that the number of plausible patches generated by any of the APR tools analysed for a given bug from the Defects4J dataset is remarkably low, a median of 2, indicating that a developer only needs to consider 2 patches in most cases to be confident to find a fix or confirming its nonexistence. This study unveils that the overfitting problem might not be as bad as previously thought. We reflect on current evaluation strategies of automated patch assessment techniques and propose a Random Selection baseline to assess whether and when using such techniques is beneficial for reducing human effort. We advocate future work should evaluate the benefit arising from patch overfitting assessment usage against the random baseline.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

Overfitting, Automated Program Repair, Patch Assessment

ACM Reference Format:
Justyna Petke, Matias Martinez, Maria Kechagia, Aldeida Aleti, and Federica Sarro. 2024. The Patch Overfitting Problem in Automated Program Repair: Practical Magnitude and a Baseline for Realistic Benchmarking. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3663529.3663776

FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663776

1 INTRODUCTION

Patch overfitting is a well-known problem in automated program repair (APR), and arguably, prevents wider adoption of APR tooling [13]. An important aspect of patch quality is whether an APR-generated patch actually fixes the given bug. APR relies on measures of correctness. Finding such a measure is an open challenge, and applies both to patches produced by humans and by machines. In APR, a patch is deemed *overfitted* if it satisfies a given APR-tool functionality criterion (usually by passing an existing test suite), yet is actually incorrect (e.g., bug prevails under untested inputs). The patch overfitting problem results in automatically generated patches that cannot be trusted by practitioners, which in turn impacts the applicability of APR [12].

In order to minimise the impact of overfitting, many approaches have been developed that either impose some constraints on the patch during the generation process, e.g., via a domain-specific language [7], or after the APR process has finished, e.g., by analysing patch behaviour on a new set of test cases [18], or based on similarity to existing code [3]. To increase the uptake of APR, these approaches aim to maximise the probability of showing correct patches to the developer, who can then decide whether to apply the patch. It is a common belief that such approaches are needed, since APR tools usually produce large amounts of plausible patches [1, 16, 18], which may have implications on the effort needed to find the correct patch. However, there is no study that verifies this assumption by actually quantifying how much overfitting exists per tool per bug. The typical empirical assessment of most APR tools often involves evaluating the accuracy of the generated patches, thereby examining the potential for overfitting. This assessment typically takes place on the Defects4J bugs. These studies run the APR tools until the first plausible patch is found, which may or may not be correct. Considering only a single plausible patch does not provide insights into the extent of overfitting, that is the likelihood of these tools to produce incorrect plausible patches. This is especially true nowadays, with increased use of neural-based approaches, that output a set of probable fixes rather than a single patch [11]. Instead, prior research on overfitting assessment of APR techniques has provided evidence of effectiveness by evaluating approaches on metrics such as accuracy, precision, and recall [14, 17, 19], or the numbers of bugs fixed vs. not-fixed, and/or total numbers of plausible yet incorrect patches generated per project [14, 19]. Although these results demonstrate the effectiveness of patch overfitting assessment approaches, **we lack an understanding of how severe the problem of overfitting is for many existing APR techniques in practice**. In particular, most post-processing approaches were evaluated on a set of previously labelled patches, without actually running the tools first. Such datasets often consolidate patches obtained in different experimental environments. Many tools stop at the first plausible patch by default, but take different amounts of time to generate one. Thus, it is not clear how many patches would be generated and whether the efficacy of the tooling at finding bugs would increase, if all tools were allowed to run for the same amount of time. As a result, we do not know how effective the post-processing techniques that tackle overfitting are at reducing the human effort in identifying the correct patch in such a scenario.

Moreover, to know if a patch overfitting assessment technique is beneficial or not, we must establish a baseline that determines the probability of selecting a correct patch from the generated patches. If the probability of selecting a correct patch is 1 (which some techniques can achieve for particular bugs), then patch overfitting assessment would not be required. We propose **all patch assessment techniques must at least beat a Random Selection (RS) strategy, i.e., the RS baseline**, — a comparison not yet done.

In this work we first pick a sample of existing results reported for automated patch assessment techniques, and calculate how a random selection strategy would fair against these. We show that for many bugs random sampling is just as effective, i.e., at least the same number of patches would have to be manually assessed to find a fix. Next, we run 10 APR tools on a set of 395 bugs from the Defects4J v1.5 [5] dataset, allowing them to run beyond the first patch found, up to a 3 hour time limit, as often used in evaluation of APR tooling [9]. We analyse the patches found and calculate the RS baseline per bug and per tool — this establishes the overfitting rate. Our study shows, for each tool, how many patches need to be sampled at random to have high confidence that a correct patch exists among generated plausible ones. This establishes a baseline for techniques that tackle overfitting that use the APR tools analysed — they must at least beat this Random Selection (RS) baseline.

Our initial study confirms that the magnitude of overfitting, is not as big as previously thought, even considering older tools. Moreover, we strongly recommend that all future patch assessment techniques should be always compared against the RS baseline.

2 OVERFITTING ASSESSMENT BASELINE

We first present a baseline to benchmark existing and future overfitting assessment approaches (i.e., approaches that aim to identify correct patches from a set of plausible ones). Our intuition is that for an overfitting assessment approach to be deemed as effective, it should at least outperform a random selection strategy. To determine this baseline, we calculate the following measure: *Given a fixed set of patches, what is the probability of selecting a desirable patch using a random selection strategy?* In this work, we focus on selecting a correct patch, but the argument follows for any other *desirable* criterion. Assuming a purely random selection process, given N patches with K *desirable* patches, the probability of selecting at least 1 desirable patch in n consecutive draws is calculated based on the Hypergeometric distribution as follows:

Table 1: ObjSim and PraPR patch ranking compared with the RS baseline on bugs for which 10+ patches were generated. 'T' means a timeout after 5 minutes. All data reported in this table is from previous work [2], but the last column contains the likely rank of a correct patch (with 80% probability) if random selection is used.

Bug	Plausible patches	Genuine fixes	Rank PraPR	Rank ObjSim	Random prob. 80%
Chart-26	100	1	17	T	81
Closure-11	15	3	1	T	6
Closure-126	12	2	5	T	7
Jsoup-42	13	1	1	T	11
Math-50	30	1	30	30	24
Mockito-5	31	1	31	31	25
Time-11	32	1	1	1	26

$$Pr(X \geq 1) = 1 - Pr(X = 0) = 1 - \frac{\binom{K}{0}\binom{N-K}{n-0}}{\binom{N}{n}} = 1 - \frac{\binom{N-K}{n}}{\binom{N}{n}} \quad (1)$$

Note that when we select only one patch at random, the formula simplifies to $1 - (\frac{N-K}{N}) = \frac{K}{N}$, i.e., we have a $\frac{K}{N}$ chance of selecting a desirable patch. If none exist, i.e., $K = 0$, the probability is 0. In the APR context, an approach that targets overfitting should select a correct patch from a set of candidate patches more often than the aforementioned baseline. In our empirical study, we focus on the situation where the APR process has finished, producing a fixed set of plausible, i.e., test-passing, patches. We aim to answer the following question: *How many patches per bug does a developer need to sample at random to establish whether a correct patch exists among plausible ones?* This will form a baseline for evaluation of any patch assessment approach that aims to overcome the overfitting problem.

3 MOTIVATION

To the best of our knowledge, no previous work on patch assessment overfitting compared against a baseline based on random sampling, which we call here RS baseline for short. To illustrate the problem let us consider results reported for some existing patch overfitting assessment techniques, and compare them with the RS baseline.

Several patch assessment techniques produce a ranking of patches, with the intention of the correct one being ranked at the top. Their effectiveness is thus measured in terms of where the correct one is ranked. In order to calculate how the RS baseline would perform in this scenario, we calculate how many patches one would need to randomly sample so that the probability of selecting a correct one is at least 80%. We chose results reported for ObjSim [2] and PraPR [4] for such an illustration. ObjSim is a standalone patch prioritisation technique, while PraPR is an APR tool that prioritises patches internally. The author of ObjSim compared against PraPR's strategy and released an artefact, containing detailed patch information per bug, which allows us to make a comparison with the RS baseline. We present these results, with an extra column for the RS baseline in Table 1[1]. Although in most cases PraPR beats the RS baseline, ObjSim does not perform as well. The comparison with the RS baseline suggests that in two cases (Mockito-5 and Math-50) a developer is better-off selecting patches at random rather than

[1]All results available at: https://github.com/SOLAR-group/overfitting-baseline-artefact/blob/main/saved_results/paper_results/table4Full.md

using PraPR. Whereas, for all cases but one in Table 1 selecting patches at random should be preferred to ObjSim.

Patches used in most patch overfitting assessment studies gathered labelled patches from previous work, and thus, they share the threat of coming from *different* experimental setups. Of course a technique that picks the one correct patch from a set of, say, 10,000, will be considered best, but the question is how would it fair *in practice*. Suppose an APR tool generates only 2 patches per bug. Perhaps it would be advantageous to use a more efficient, but maybe less effective automated patch assessment technique, if it can correctly decide if they are overfitted with high probability? Perhaps random selection or even manual effort is less time-costly? We do not aim to answer these questions here, but instead, provide a set of patches generated using a scenario where each tool can generate patches up to the same time limit, and not just stop at first patch found. We show the rate of overfitting per tool and bug, with calculations of the effectiveness of a random patch selection strategy. Our work provides a baseline for future studies using this dataset.

4 METHODOLOGY

We considered 10 APR tools (Table 2) widely studied (e.g., [10]) and evaluated by state-of-the-art patch assessment approaches such as Cache [8] and ODS [19]. We modified them, if needed, to run beyond the first patch found. We use the widely studied Defects4J v1.5 [5] dataset, due to the availability of labelled patches. We ran each tool with the same time limit and report on patches obtained within the first 3 hours, as standard in previous studies [9]. We examined all plausible patches found in order to determine their correctness based on four different subsequent assessments: syntax comparison with developer-written patches, with patches from previous work [6, 8, 15], automated dynamic analysis with extra tests [20], and manual analysis by two independent reviewers of leftover unlabelled patches. Due to time constraints, we performed the last two analysis stages (dynamic and manual) for the patches for bugs for which a correct patch was found during syntactic checks. We ran each repair attempt on an Intel Xeon E5-2630 v3 (Haswell, 2.40GHz, 2 CPUs, 8 cores/CPU) and 128 GB RAM.

5 RESULTS

The 10 APR tools generated 4,643 patches, 2,304 of which unique. 155 bugs were plausibly patched. Through syntactic analysis we identified 43 bugs for which a correct patch was found.

Magnitude of Overfitting Interestingly, for all 10 tools, the median number of unique patches generated for the 155 bugs (for which plausible patches have been generated by at least one tool) is 2, while the average is 6.71. This is already a striking result. If there are only 2 patches to verify, this puts into question the need for sometimes quite time-consuming automated patch overfitting approaches, especially if they can produce false-negative results, i.e., mark a correct patch as an overfitting one. For the 10 APR tools and 43 identified correctly fixed bugs the median number of patches generated per tool per bug is 2, while the average is 4.11, with a max of 45. Indeed, on this dataset for a single tool for a single bug either no plausible patches are generated or, most frequently, just 2 plausible patches are produced. This means that unless a post-APR overfitting approach is very quick and easy to use, it might be more cost-effective for a developer to manually analyse the two patches.

Table 2: Number of patches n (Equation 1) per tool to be sampled to have 80%, 90%, or 100% chance of selecting a correct patch at random. Data per bug is aggregated per tool.

		Sample n sufficient with x% probability					
	Bugs	median			maximum		
Tool	fixed	80%	90%	100%	80%	90%	100%
Avatar	13	1	1	1	5	6	6
FixMiner	9	1	1	1	3	3	4
Nopol	0	0	0	0	0	0	0
SimFix	19	1	1	1	3	3	3
TBar	25	1	1	1	23	28	40
dynamoth	2	2	2	4	3	3	6
jGenProg	3	2	2	2	3	3	3
jKali	2	4	5	6	5	6	6
jMutRepair	4	2	2	2	3	3	3
kPAR	5	3	3	3	7	10	37

Overfitting Per Tool Table 2 presents the number of patches a developer using a particular APR tool would have to randomly sample to have 80%, 90%, 100% chance of selecting a correct patch for a given bug, by using the RS baseline (see Section 2). A median value of 1 or 0 means that for most bugs the patches generated by a given tool are either all correct or incorrect, respectively. We note that Nopol did not generate any correct patches for any of the bugs for which at least one other tool generated a correct patch. It is worth noting that in previous section only a median of 2 patches was generated per bug. Consequently, even in situations where the likelihood of choosing the correct patch is 0 due to overfitting, the patch assessment process still only requires assessing these 2 patches.

For half of the tools, the highest number of patches that must be sampled to achieve a 90% likelihood of discovering a correct patch (assuming one exists) for a given bug is 3 (refer to the second-to-last column in Table 2). More interestingly, SimFix and TBar, tools that are able to fix the largest numbers of bugs, require max 3 and 28 patches to be sampled, respectively, while the median is 1 for both. Hence, if a developer intends to employ either SimFix or TBar, they could randomly select just one patch. In the majority of instances, this single selection would provide sufficient information to determine if a correct patch has been generated. Additionally, in the case of SimFix, they might contemplate evaluating up to three different patches. For 8 tools only up to 5 incorrect patches have been generated per bug. This means that if one samples any 6 patches per bug generated by any of these tools, one would have certainty whether a correct patch was generated or not. TBar, on the other hand, generated 41 unique patches for Math-85, 2 of which were correct.[2] It is worth noting that jMutRepair generated only 3 patches for this bug, one of them correct. kPAR generated 45 unique patches for Math-50, 9 of which correct. It was, however, the only tool to generate a correct patch for this bug.

Our results also reveal that the more recent tools are more effective at finding correct patches, such as SimFix and TBar (Table 2). In particular, TBar is able to find the largest number of fixes, i.e., 25, for 43 bugs for which known fixes were found by any of the 10 tools. Each of those bugs was fixed by the 1st plausible patch

[2] https://github.com/SOLAR-group/overfitting-baseline-artefact/blob/main/saved_results/paper_results/rq2/preprocessed.csv

Table 3: For each tool we show the no. of bugs fixed if the first n patches are considered per bug (for selected n).

	No. of bugs fixed within the first n=1,2,..,37 plausible patches per bug								
Tool	1	2	3	4	5	20	21	36	37
TBar	27	29	29	29	30	31	32	32	32
SimFix	19	19	19	19	19	19	19	19	19
jMutRepair	3	3	4	4	4	4	4	4	4
Avatar	12	14	14	14	14	15	15	15	15
kPAR	4	4	4	4	4	4	5	5	5
FixMiner	9	10	10	10	10	10	10	10	10
jGenProg	2	2	2	2	2	2	2	2	3
dynamoth	2	2	2	2	2	2	2	2	2
jKali	1	1	1	1	2	2	2	2	2

Table 4: No. of patches n (Equation 1) per bug to be sampled, from the set of all patches generated by 10 tools, to have 80%, 90%, or 100% chance of selecting a correct patch at random.

	Sample n sufficient with x% probability						
Bug	80%	90%	100%	Bug	80%	90%	100%
Chart-11	1	1	1	Lang-57	3	3	4
Chart-20	1	1	1	Math-70	3	3	4
Chart-24	1	1	1	Math-79	4	4	4
Chart-4	1	1	1	Mockito-29	4	4	4
Chart-8	1	1	1	Time-7	4	4	5
Closure-14	1	1	1	Math-30	4	5	6
Closure-73	1	1	1	Math-82	5	6	7
Closure-86	1	1	1	Chart-1	3	4	8
Math-11	1	1	1	Closure-46	5	6	9
Math-34	1	1	1	Lang-55	3	4	9
Math-5	1	1	1	Chart-9	9	10	11
Math-59	1	1	1	Closure-62	5	6	12
Math-65	1	1	1	Closure-126	6	8	15
Math-75	1	1	1	Math-58	15	17	18
Math-89	1	1	1	Chart-7	16	18	19
Mockito-38	1	1	1	Lang-59	21	24	26
Math-57	2	2	2	Math-33	16	21	37
Closure-2	3	3	3	Math-80	32	36	39
Closure-57	2	3	3	Lang-58	35	39	43
Lang-33	2	2	3	Math-85	21	27	48
Lang-43	2	3	3	Math-50	11	15	61
Math-53	3	3	3				

found by TBar. This is not always the case. We gathered data for all tools for an 8-hour time limit, and present in Table 3. TBar generated a correct patch for 32 bugs: 27 were correctly fixed by the 1st plausible patch found, 2 more by the 2nd plausible patch found, while all 32 if one considered the 1st 21 plausible patches per bug.

Overfitting Per Bug Next, we consider the case where for a given bug we gather all syntactically unique patches generated by all 10 tools. This should increase the chances of finding a correct patch, since there are bugs for which only one tool might be able to generate a correct patch. However, this strategy might make the selection of a correct patch more difficult, as a developer would have to undertake patch assessment for a larger set of patches than if they were to use just one tool. Therefore, we calculate the probabilities of randomly sampling a correct patch from the set of all patches generated by all 10 tools for a given bug, based on the RS baseline. Table 4 reports the number of patches that need to be sampled to have 80%, 90%, or 100% chance that among them there is a correct patch. We observe that for half of the bugs (22) up to 2 incorrect patches are generated, and thus one only needs to sample 3 patches to have 100% confidence that a correct patch exists in the sampled set. However, even up to 61 plausible patches would have to be reviewed for one bug (see Math-50 in Table 4), rendering a post-APR overfitting approach useful in those cases.

Nevertheless, the results show that indeed combining patches from different tools significantly increases the number of bugs fixed, from 25 fixed by an individual tool, to 43 when 10 tools are used. This comes at the cost of a higher number of patches to be assessed, yet for 26 bugs only 3 patches need to be assessed to have 80% confidence that a correct one would be found (see 2nd column in Table 4), with the number increasing to 34 bugs if up to 6 patches are analysed per bug (and 33 with 90% confidence).

Patch Sampling Strategy Let us assume that the APR tools are used to fix bugs that are similar to the ones reported here, and thus the numbers in Tables 2 and 4 are representative of the ability of the tools to find patches for such a set of bugs. If a developer where to use all 10 tools to find a patch with 80% confidence for one bug, they would have to sample $(5 * 3 + 23 + 7 + 2 * 5) = 55$ patches, if figures from Table 2 in the 6th column, are considered, or up to 35, based on the results reported in Table 4 (in 2nd column). This is a worst-case scenario for this dataset and our chosen probability measure. However, since the developer knows which patches come from which tool, they might consider evaluating patches from SimFix in the first instance, as they only need to consider 3 of those patches in order to have confidence that a correct patch was found by the tool.

Whether it is best to consider the patches of individual tooling or combine patches together to sample from will depend on the choice of tools and bugs used, thus we advocate both sampling strategies are considered. We acknowledge that the results will differ, depending on the bugs and tools used. However, future research on post-APR overfitting that uses the same ones as us here should take into account the random empirical baselines established here.

6 CONCLUSIONS

Our empirical study confirms that ARP tools suffer from overfitting yet this problem might be not as bad as previously thought. We found that a developer only needs to consider 2 patches for most bugs to be confident to find a fix (or be sure none exists) for the Defect4J bugs considered in our study. We found that using multiple APR tools at the same time does not make the APR overfitting problem much worse. Therefore, we recommend to always evaluate the benefit arising from the use of any post-processing APR tooling. We outlined a baseline, based on Random Selection (RS), for patch overfitting assessment. The RS baseline helps us explore and analyse the benefits of post-APR overfitting assessment techniques, and when it pays off to use such techniques. **All of our scripts and labelled patches are available** at: https://github.com/SOLAR-group/overfitting-baseline-artefact. **Funding** We thank Ramony Cajal Fellowship no. RYC2021-031523-I, ERC Advanced Grant no. 741278, UKRI EPSRC grant no. EP/P023991/1, Australian Research Council grant no. DP210100041.

REFERENCES

[1] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical review of Java program repair tools: a large-scale experiment on 2, 141 bugs and 23, 551 repair attempts. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 302–313. https://doi.org/10.1145/3338906.3338911

[2] Ali Ghanbari. 2020. ObjSim: Lightweight Automatic Patch Prioritization via Object Similarity. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 541—-544. https://doi.org/10.1145/3395363.3404362

[3] Ali Ghanbari and Andrian Marcus. 2022. Patch correctness assessment in automated program repair based on the impact of patches on production and test code. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 654–665. https://doi.org/10.1145/3533767.3534368

[4] Ali Ghanbari and Lingming Zhang. 2019. PraPR: Practical Program Repair via Bytecode Mutation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1118–1121. https://doi.org/10.1109/ASE.2019.00116

[5] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[6] Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. 2019. Effective and efficient API misuse detection via exception propagation and search-based testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 192–203. https://doi.org/10.1145/3293882.3330552

[7] Xuan-Bach Dinh Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 593–604. https://doi.org/10.1145/3106237.3106309

[8] Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. 2022. Context-Aware Code Change Embedding for Better Patch Correctness Assessment. *ACM Trans. Softw. Eng. Methodol.* 31, 3 (2022), 51:1–51:29. https://doi.org/10.1145/3505247

[9] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *J. Syst. Softw.* 171 (2021), 110817. https://doi.org/10.1016/j.jss.2020.110817

[10] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, New York, NY, USA, 615–627. https://doi.org/10.1145/3377811.3380338

[11] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 101–114. https://doi.org/10.1145/3395363.3397369

[12] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2228–2240. https://doi.org/10.1145/3510003.3510040

[13] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 532–543. https://doi.org/10.1145/2786805.2786825

[14] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. 2020. *Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair.* Association for Computing Machinery, New York, NY, USA, 981–992. https://doi.org/10.1145/3324884.3416532

[15] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated Patch Correctness Assessment: How Far Are We?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 968—-980. https://doi.org/10.1145/3324884.3416590

[16] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 789–799. https://doi.org/10.1145/3180155.3180182

[17] Jun Yang, Yuehan Wang, Yiling Lou, Ming Wen, and Lingming Zhang. 2023. A Large-Scale Empirical Review of Patch Correctness Checking Approaches. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1203–1215. https://doi.org/10.1145/3611643.3616331

[18] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 831–841. https://doi.org/10.1145/3106237.3106274

[19] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2022. Automated Classification of Overfitting Patches With Statically Extracted Code Features. *IEEE Trans. Software Eng.* 48, 8 (2022), 2920–2938. https://doi.org/10.1109/TSE.2021.3071750

[20] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated patch assessment for program repair at scale. *Empir. Softw. Eng.* 26, 2 (2021), 20. https://doi.org/10.1007/s10664-020-09920-w

Received 26-JAN-2024; accepted 2024-04-09