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

Justyna Petke
University College London
London, United Kingdom

Matias Martinez
Universitat Politècnica de Catalunya-BarcelonaTech
Barcelona, Spain

Maria Kechagia
University College London
London, United Kingdom

Aldeida Aleti
Monash University
Melbourne, Australia

Federica Sarro
University College London
London, United Kingdom

Abstract

Automated Program Repair techniques aim to generate patches for software bugs, mostly relying on testing to check their validity. It is possible that such a *plausible* patch may not be correct, as there still might exist inputs under which the bug prevails.

We carry out a thorough empirical study to analyse the extent to which program repair tools suffer from this *overfitting problem* in practice. We present how a random baseline can be used to understand whether and when the use of automated methods is beneficial for reducing human effort in patch assessment.

We find that the number of plausible patches generated by any of the program repair tools analysed for a given bug from the Defects4J dataset is low — a median of 2. The median number of patches one needs to randomly sample to have high confidence that a correct patch exists within the generated patch set using 10 program repair tools for each of the bugs is only 6 at most. Moreover, for most tools only 3 plausible patches are generated before producing a correct one.

1 Introduction

Patch overfitting [Smith et al.(2015)] is a well-known problem in automated program repair (APR), and arguably, prevents a wider adoption of APR tooling. 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 [Noller et al.(2022)].

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 [Le et al.(2017)]) or after the APR process has finished, e.g., by analysing patch behaviour on a new set of test cases [Yang et al.(2017), Xiong et al.(2018)]. 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 tens or even hundreds of plausible patches [Yang et al.(2017), Xiong et al.(2018), Durieux et al.(2019a)], which may have implications on the effort needed for finding the correct patch. However, to the best of our knowledge, there is no study that verifies this assumption by actually quantifying how much overfitting exists per tool per bug in a practical scenario.

Prior research on overfitting assessment of APR techniques has provided evidence of effectiveness by evaluating the approaches on metrics such as accuracy, precision, and recall [Ye et al.(2021a), Tian et al.(2020)], or the numbers of bugs fixed vs. not-fixed, and/or total numbers of plausible yet incorrect patches generated per project [Ye et al.(2021a), Tian et al.(2020)]. For example, Patch-sim [Xiong et al.(2018)] is able to exclude up to 69.2% of incorrect patches

(9 out of 13 incorrect patches for bugs from the Time project in Defects4J), while correctly classifying all correct patches. While these results are impressive and demonstrate the effectiveness of the approaches for patch overfitting assessment, we lack an understanding of how severe the problem of overfitting is for the 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. 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 would increase (by fixing more bugs), 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 an arguably more fair 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 previous work.

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 [Just et al.(2014)] dataset, allowing them to run beyond the first patch found, up to an 8 hour time limit — this shows if tool effectiveness increases with time. We analyse patches found within the first 3 hours, using this limit as often used in evaluation of APR tooling [Liu et al.(2021)], and calculate the RS baseline per bug and per tool — this establishes the overfitting rate in a practical scenario. 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 experiment provides several insights:

- The median number of patches generated per tool per bug is 2 — developers only need to review 2 patches to establish correctness in most cases (RQ1);
- one needs to randomly sample up to 6 unique patches per bug to have high confidence that a correct patch has been generated or not (RQ2), and can use patches from multiple tools to do so — developers can disregard patch assessment of more than 6 randomly sampled patches per bug (if they use the 10 tools we analyse) (RQ3) ;
- knowledge of which tool generated a patch can speed-up patch assessment (RQ4);
- fixes for 91% (96%) of bugs are generated with the first (second) plausible patch if multiple tools are used, with 21 patches needed to fix all bugs within 4.5 hours — generation beyond first plausible patch can further find fixes (RQ5).

Overall, we recommend that developers should 1) use multiple APR tools with the same time limit 2) assess only up to 6 randomly selected patches, taking the tool type into account. Furthermore, we strongly recommend that all future patch assessment techniques, should be always compared against the RS baseline.

To summarise, our contributions are as follows:

- We present a baseline, based on random sampling, for approaches that target overfitting in APR. This baseline indicates the minimum conditions that must be met in APR for an overfitting assessment approach to be regarded effective.
- We conduct a large-scale empirical study to examine the magnitude of overfitting per APR tool per bug. This enables us to analyse the need for patch assessment, and whether particular tools or bugs are more prone to overfitting.
- We provide practical recommendations to software developers (listed above).
- We provide a dataset of patches, generated by 10 APR tools in a practical scenario. Each tool was run for the same amount of time, beyond the first patch found. We provide RS baselines for the 3-hour time limit.

2 Overfitting Assessment Baseline

In this section, we 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:

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

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 [Ghanbari(2020)] and PraPR [Ghanbari and Zhang(2019)] 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.[1] We present these results, with an extra column for the RS baseline in Table 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 using PraPR. Whereas, for all cases but one (where there are at least 10 plausible patches) selecting patches at random should be preferred to ObjSim.

Another way to assess patch overfitting assessment efficacy is to calculate the reduction from a set of plausible patches generated for a given bug [Wang et al.(2020)], and/or provide precision/recall for patch correctness prediction [Lin et al.(2022)]. Most recently, Cache [Lin et al.(2022)] was proposed to predict patch correctness, with the authors having released a dataset of labelled patches. For instance, let us consider Cache's results for the Chart-1 bug.[2] The dataset contains 3 correct and 14 overfitted patches for this bug. Cache does not predict any of the correct patches to be correct. Thus, a developer would be advised that no correct patches were found. If instead a developer were to pick 6 of those at random for evaluation, they would have had 76% chance of picking at least one correct patch.[3] In fact, Cache incorrectly predicts that for 100 out of 356 bugs no correct patch exists in their dataset, while with random

[1] Subset shown in Table 1. Full results available here: `https://github.com/SOLAR-group/overfitting-baseline-artefact/tree/main/saved_results/paper_results/table4Full.md`

[2] `https://github.com/Ringbo/Cache/blob/master/Results/RQ1/Small/Cache.csv`

[3] Result from equation in Section 2 with $K = 3$, $N = 17$, and $n = 6$.

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 [Ghanbari(2020)], 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

selection strategy for 84 of those the developer would only need to randomly select up to 6 patches to find a fix with 80% chance.[4]

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 *practical* scenario. We show the rate of overfitting in practice per tool and bug, with calculations of the effectiveness of a random patch selection strategy. Our work provides a baseline for future studies.

4 Research Questions

Our study has two main goals. Firstly, we aim to establish the extent to which APR tooling suffers from overfitting in practice. Secondly, our goal is to provide a baseline for the evaluation of approaches that tackle APR overfitting that the community could use to evaluate current and future proposals.

In an ideal scenario, an APR tool will only produce correct patches, or none, if a correct patch cannot be found. However, it is currently believed that APR tools mostly produce incorrect patches, and thus would require a lot of human effort for patch assessment of the generated patches [Ye et al.(2021a), Wang et al.(2020)]. Moreover, even if a correct patch exists within all the plausible patches generated for a given bug, the effort required for their evaluation might not be worth it. In fact, that effort could be better spent by the developer on manual analysis of code to derive a patch from scratch. Furthermore, many of the APR tools in practice are set to stop at the first plausible patch found, and, thus, we do not know *what would be the efficacy of a given APR tool if we let it run for longer?* (i.e., *would it find a correct patch?*), and *how many more patches would be produced for a given bug?*. Answering these questions would let us better understand the extent to which current APR tools suffer from the overfitting problem. Our first research question aims to shed more light on this matter:

> RQ1: *Magnitude of Overfitting*. How many patches are generated if the APR process does not halt at the first plausible patch found?

To answer RQ1, we run a set of APR tools with the same stopping condition (i.e., we do not halt them at the first plausible patch, but rather we let them execute for the same amount of time) and gather all patches generated for a given bug (see Section 5). We label the generated patches as correct ones if they syntactically match known fixes. We use automatically generated tests for a fix to filter out overfitting patches for those bugs for which a fix was found. After this step, we manually label plausible patches left. Then, we compute and report on the number of patches found and their type to understand the magnitude of the APR overfitting problem for each of these tools, and overall across all tools per bug.

Even if we found that current APR tooling would generate a very large number of plausible patches per bug, it would be useful to investigate whether a given tool only ever generates correct or incorrect patches per bug. If this is the case,

[4]Full result table: `https://github.com/SOLAR-group/overfitting-baseline-artefact/tree/main/saved_results/paper_results/cache_comparison.csv`

a developer would only need to sample one patch at random to establish whether a correct patch was generated or not. This motivates our second question:

> RQ2: *Overfitting Per Tool.* How many patches a developer needs to randomly sample per tool per bug to establish whether a correct patch was generated?

To answer RQ2, we focus on the patches generated by each tool for each bug. We analyse those bugs for which a tool generated at least one correct patch over the same execution time limit. For each pair of bug and tool, we analyse the probability of sampling at least one correct patch per set of generated patches.

Different APR tools might exhibit different effectiveness for a certain bug [Liu et al.(2020), Aleti and Martinez(2021)]. Therefore, in order to increase chances of finding a correct patch, a developer might choose to run multiple APR tools rather than relying only on a specific one. This will consequently increase the number of patches generated, and potentially the effort required to alleviate overfitting, yet there might be cases where all tools are able to find a correct patch, or only one tool is able to generate one patch. Similarly to RQ2, analysing what happens in practice when APR tools do not halt at the first plausible patch found would help us gain a better understanding of the overfitting problem. Therefore, our third research question asks:

> RQ3: *Overfitting Per Bug.* How many patches a developer needs to randomly sample from all available APR tools to establish whether a correct patch was generated for a given bug?

To answer RQ3, we aggregate all the patches generated by the APR tools for a particular bug over the execution time limit and carry out the same analysis as for RQ2.

When we gather together the patches produced by each APR tool the total number may add up to hundreds of patches. Analysing which tool generated which patches might help us derive some insights on whether there are tools from which one should sample first. For example, if we find that certain APR tools always generate either all correct patches or incorrect ones, then a developer would need to sample only one patch generated by such a tool to determine if a correct patch exist. Therefore, our fourth research question asks:

> RQ4: *Patch Sampling Strategy* When evaluating patches generated by multiple tools for a given bug, is it better to sample patches per tool or from the whole set of patches generated?

To answer RQ4 we consider the patches generated by each tool per bug over the same execution time limit. We compare the strategy of random sampling from patches generated per tool vs. randomly sampling from the set of all patches generated by all tools. If, for instance, one tool generates 2 correct patches, while another 100 incorrect ones, then sampling from the set of patches generated by the first tool is clearly more beneficial than sampling from the whole set of 102 patches for a given bug.

Finally, we would like to understand the extent to which an APR tool can generate a correct patch if it does not halt at the first plausible patch found. To this end, our fifth research question investigates for how long each of the APR tools should run in order to find a correct patch:

> RQ5: *APR Tool Usage.* How long shall an APR tool run for? How many plausible patches are needed before one can have confidence that the tool is (or is not) able to find a correct patch?

To answer RQ5, we analyse the patch generated by each of the tools over a longer execution time (8 h). This allows us to understand if and when a correct patch has been found.

5 Methodology

In this section, we present the methodology that we follow to answer the research questions posed in Section 4. Figure 1 shows the steps we undertook in our experiments. We first generate patches by executing several existing APR approaches for the same amount of time, as detailed in Section 5.1. Then, we examine all plausible patches found by these tools in order to determine their correctness based on four different subsequent assessments (i.e., based on developer-written patches, on previous work, on automated analysis, and on manual analysis), as explained in Section 5.2. Finally, we analyse this data to respond to each of our research questions to gain understating of the overfitting problem per tool and per bug.

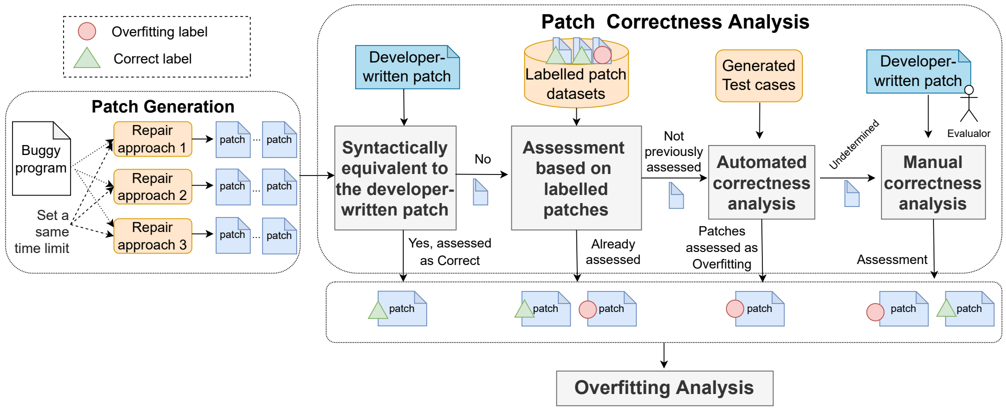

Figure 1: Overview of the methodology adopted to generate and label the patches obtained by using multiple APR approaches.

5.1 Patch Generation

In this section, we detail the procedure we adopt to generate patches by using existing APR tools. We describe the corpus of bugs, and the APR tools and settings, used in our empirical study, as well as the rationale behind our choices.

5.1.1 Bugs

In this work, we focus on repairing Java bugs because: *1)* Java is one of the most adopted programming languages in the industry. *2)* APR approaches have been widely evaluated on Java programs (e.g., [Liu et al.(2019b), Liu et al.(2018), Jiang et al.(2018), Lutellier et al.(2020)]). *3)* Overfitting analysis and approaches have mainly focused on the analysis of Java patches (e.g., [Xiong et al.(2018), Ye et al.(2021a), Wang et al.(2020), Lin et al.(2022)]). We use the Defects4J v1.5 [Just et al.(2014)] dataset, which contains 395 bugs that have been widely used in the assessment of the recently proposed Java repair systems (e.g.,[Koyuncu et al.(2020), Lutellier et al.(2020), Chen et al.(2021), Jiang et al.(2021)]) as well as by other studies evaluating existing approaches [Martinez et al.(2017), Durieux et al.(2019b), Liu et al.(2020)]. Using Defects4J allows us to leverage on the information, data and knowledge built in previous studies, which is relevant to carry out our own experiments and to facilitate future replications, extensions, and comparisons. In particular, we can reuse from such previous work: *a)* labels for patches generated by other studies that have been manually or automatically assessed, and *b)* tools devised for automatically assessing Defects4J bugs.

The reuse of such information is valuable to us because it allows us to: *1)* speed up patch correctness analysis (we do not have to assess again patches already assessed), and *2)* reduce the threats to the validity of the patch analysis step [Ye et al.(2021b), Wang et al.(2020)].

5.1.2 APR Candidate Tools

We considered an initial set of 16 APR tools, which has been previously used by Liu et al. [Liu et al.(2020)] since these tools are all open-source and working [Liu et al.(2020)], and capable of repairing bugs from Defects4J: Avatar [Liu et al.(2018)], ARJA [Yuan and Banzhaf(2018)], ACS [Xiong et al.(2017)], Cardumen [Martinez and Monperrus(2018)], Dynamoth [Durieux and Monperrus(2016)], FixMiner [Koyuncu et al.(2020)], GenProg-Arja [Yuan and Banzhaf(2018)], jGenProg [Martinez and Monperrus(2016)], jKali [Martinez and Monperrus(2016)], jMutRepair [Martinez and Monperrus(2016)], kPAR [Liu et al.(2019a)]. Kali-ARJA [Yuan and Banzhaf(2018)], Nopol [Xuan et al.(2017)], RSRepair [Martinez and Monperrus(2016)], SimFix [Jiang et al.(2018)] and TBar [Liu et al.(2019b)]. These tools are all publicly available on GitHub, so we downloaded the latest version of each tool, at the time of the experiment, by cloning the default branch. We then inspected each of the tools in order to check if they satisfy our study's requirements in terms of the execution timeout and stopping condition. Specifically, we need to use APR tools that are able to: *1)* Continue analysing the search space after finding the first patch that passes all the tests contained in the buggy program. We recall that those are called *plausible* patches. *2)* Execute for a specified amount time. To this end, we carefully inspected the source code of each of the 16 tools.

5.1.3 Enabling Multiple Plausible Patches Generation and Fixed Execution Time.

In order to collect more than one plausible patch and enable the execution of the tools for a specific amount of time, we had to modify these tools that do not support the preferred options by default. We thus modified the source code of ARJA (including Arja, GenProg-Arja, RSRepair-Arja, Kali-Arja), Nopol (including Dynamoth), SimFix, TBAR, AVATAR, kPAR and FixMiner to add two command-line arguments that indicate: *a)* if the tool must continue or stop after the first plausible patch found, *b)* the timeout in minutes. We also modified each of the tools to register and report the time that it takes to find each plausible patch. The modified versions of the tools are available in our repository. [5]

To assure the correct behaviour of a given tool after our modifications, we executed the modified version on three Defects4J bugs that were repaired by the original (not modified) tool, and verified if the first patch found was the same.

After this step, we ended up with 16 repair tools successfully modified by us, which are able to generate multiple patches over a specific period of time, which can be passed as a parameter to each tool.

5.1.4 Running APR tools

We then executed the modified version of each tool on each of the Defects4J bugs. We use the Defects4J version 1.5, which contains 395 bugs. We call *repair attempts* each of those executions. Each repair attempt is executed once for 8 hours and each tool continues after finding the first plausible patch. A tool can stop before the timeout only if it happens to completely navigate its search space (this can happen for example when a tool uses a set of mutations for a particular construct, e.g., an `if` statement, then it only has a fixed number of possibilities to explore, and therefore the search space of patches could be exhaustively evaluated). It is important to remark that we do not use any other argument beyond those required for minimal usage (i.e., each tool uses the *default* values defined by their developers).

We ran each repair attempt on a Dell PowerEdge R630 machine, with Intel Xeon E5-2630 v3 (Haswell, 2.40GHz, 2 CPUs, 8 cores/CPU), and 128 GiB of RAM memory.

5.1.5 APR Tools Evaluated

Among the 16 tools we analysed, we could evaluate results obtained from 10 in our empirical study: Avatar [Liu et al.(2018)], FixMiner [Koyuncu et al.(2020)], Nopol [Xuan et al.(2017)], SimFix [Jiang et al.(2018)], TBar [Liu et al.(2019b)], Dynamoth [Durieux and Monperrus(2016)], jGenProg [Martinez and Monperrus(2016)], jKali [Martinez and Monperrus(2016)], jMutRepair [Martinez and Monperrus(2016)] and kPAR [Liu et al.(2019a)]. We had to exclude the other six (GenProg-Arja [Yuan and Banzhaf(2018)], RSRepair [Martinez and Monperrus(2016)], Kali-ARJA [Yuan and Banzhaf(2018)], ARJA [Yuan and Banzhaf(2018)], ACS [Xiong et al.(2017)], Cardumen [Martinez and Monperrus(2018)]) for reasons explained below.

As our experiment is demanding both in terms of computational resources — for carrying out automated patch assessment (see Section 5.2.3) — and of human resources — for carrying out manual patch assessment (see Section 5.2.4), we decided to simplify our experiment by choosing only one tool per approach whenever the same approach is implemented by multiple tools. This allows us to save computational and human resources by avoiding the analysis of duplicated patches produced by two tools that implement the same approach. In our case, there are two approaches that are implemented by more than one tool: GenProg [Le Goues et al.(2011)] by GenProg-Arja and jGenProg, and Kali [Qi et al.(2015)] by Kali-Arja and jKali. We choose jGenProg and jKali for the following reason. After the execution of those tools, we find that the approaches from Arja generate a large amount of patches, much larger than any other tool. For example, GenProg-Arja generates 296,623 patches, probably due to incorrectly reporting also non-plausible patches. Such a large amount of patches makes infeasible both manual and automated patch analyses. For that reason, we discard from our study GenProg-Arja and Kali-Arja tools, and use jGenProg and jKali to represent GenProg and Kali, respectively. Moreover, following the recommendation from ACS's authors written in the ACS repository,[6] we use SimFix [Jiang et al.(2018)] instead of ACS. At this step we thus excluded 3 tools from our experiments (i.e., ACS, GenProg-Arja, and Kali-Arja).

Next, we collected all plausible patches generated by each of the 13 tools in the 8 hour time limit. At this point, one needs to assess all the plausible patches in order to determine whether they are correct (i.e., *patch assessment*). Overall, 218,552 patches were generated, 186,277 syntactically unique. We noticed that three tools (ARJA, RSRepair, and Cardumen) produced significantly more patches than all the others: ARJA over 163k, RSRepair 20k, Cardumen 1.5k.

[5] Source code of the modified tools `https://github.com/SOLAR-group/overfitting-baseline-artefact/tree/main/tools/`

[6] ACS repository: `https://github.com/Adobee/ACS`, last visit March 17 2022.

Such large numbers of patches makes it infeasible to conduct manual analysis, and thus these tools will unlikely be used in practice. In contrast to previous studies, we ran the tools for 8 hours each, and we do not stop the search at the first plausible patch found. We observed that those tools may be affected by a bug that reports either implausible patches or non-minimised patches (spurious changes not related to the bugfix). The fact that previous work [Liu et al.(2020), Durieux et al.(2019b)] halts at the first patch found did not fully expose the problem. Therefore, since we focus on a practical scenario, and these tools will unlikely be used in practice, we discarded all patches produced by them.

This leaves us with 5,454 plausible patches (of which 2,841 unique) to analyse. Strictly speaking, the analysis presented to answer RQs 1–4 is based on the set of plausible patches generated in the first 3 (of the 8) hours, while to answer RQ5 we provide insights on all plausible patches generated over 8 hours. Full results for the two time limits are included in our on-line repository. Overall, the 10 tools generated patches for 158 bugs, for 155 of which within 3h.

5.2 Patch Assessment

The patch assessment stage takes as input the patches generated by the APR tools and outputs, for each of them, a label indicating whether a patch is *Correct* or *Overfitting* (i.e., incorrect).

We break this task into four different steps: (1) assessment based on developer-written patches, (2) assessment based on labels obtained from previous work; (3) assessment based on automated analysis using extra test cases; (4) assessment based on manual analysis). Each assessment is described in the following subsections.

5.2.1 Equivalence to Developer-written Patches

The first assessment consists of determining if a generated patch for a given bug is syntactically equivalent to the patch written by the developer to fix that bug. Defects4J provides the developer-written patch for each bug. The syntactic equivalence check between two patches is carried out by doing `String` comparisons (ignoring white-spaces, empty lines). If a generated patch is equivalent to the developer-written, we assess it as correct. Otherwise, the patch is assessed in the next step.

5.2.2 Previous Correctness Annotations

The second assessment is based on annotations made in previous work. In this study, we reuse the annotations produced in four previous studies by either manual analysis [Wang et al.(2020), Kechagia et al.(2019)] or automated techniques [Ye et al.(2021b)], or gathered from previous work [Lin et al.(2022)]. For instance, Liu et al. [Liu et al.(2020)] and Kechagia et al. [Kechagia et al.(2019)] manually classified patches as correct or incorrect generated by different existing repair tools. Then, Wang et al. [Wang et al.(2020)] found a few misclassifications in the dataset of Liu et al. [Liu et al.(2020)]. Therefore, we considered the correct dataset produced by Wang et al. [Wang et al.(2020)]. He et al. [Ye et al.(2021b)], on the other hand, automatically labelled generated patches (collected from previous work) by using test cases automatically generated based on human-written patches. Lin et al. [Lin et al.(2022)] collected the largest set of labelled patches for Defects4J to-date from previous work. We use it for syntactic analysis as well. If a patch generated by our experiment was already labelled by any of the aforementioned previous work, we reused the labels that these works assigned to the same patch. This check was done by an automatic syntactical comparison of patches, same as described for developer-ones. To ensure the reliability of this labelling process, two of the authors manually analysed all the patches labelled as correct found in previous work. The results of this analysis are in our online replication package (Section 10).

Based on the above process we labelled 226 of the patches generated in our experiment. This left us with 2,615 unique patches still to be assessed. Since the probability of selecting a correct patch for a bug for which there are only overfitted patches is 0, we focus our labelling efforts only on the patches for those bugs, for which a correct patch has already been found. According to our syntactic check, we can find correct patches for 45 bugs. Thus, we further analysed the patches for those bugs only.

For the 45 bugs for which a correct patch was already identified, the 10 tools generated 1,223 patches, 572 of which were unique, with 110 already labelled through our syntactic checks. This leaves us with 462 unlabelled patches. We present the evaluation of these patches in the next step.

5.2.3 Automated Assessment

We executed the automated labelling approach *RGT*, proposed by He et al. [Ye et al.(2021b)]. RGT generates new test cases from the patched version of the program, patched with the correct ground-truth patch, written by the developer. Then, the approach executes those generated test cases on a buggy program patched with a generated patch: If a

Table 2: Patches analysed, generated in 8 hours in our experiment, by 10 APR tools, for 158 bugs from Defects4J for which at least one tool generated a patch. For 45 bugs a fix was found based on syntax check with previous work. Last row shows numbers for the first 3 hours.

Stage	Bugs patched	All patches	Unique patches	Unique labeled	Unique unlabeled
Generated (5.1)	158	5,454	2,841	0	2,841
Automated syntactic check (5.2.2)	158	5,454	2,841	226	2,615
Automated syntactic check w/correct (5.2.2)	45	1,223	572	110	462
Automated test assessment (5.2.3)	45	1,223	572	275	222
Manual analysis (5.2.4)	45	1,223	572	572	0
Within first 3h	43	1,086	472	472	0

generated test case fails, it means that the generated patch has a different behaviour from the ground-truth patch, i.e., the generated patch is overfitted.

RGT includes sets of test cases generated by Evosuite [Fraser and Arcuri(2011)] and Randoop [Pacheco and Ernst(2007)] from the fixed version of the Defects4J bugs. We used two sets of test cases included in RGT: *a)* one from Shamshiri et al. [Shamshiri et al.(2015)], and *b)* a second one from He et al. [Ye et al.(2021b)]. In our experiment, we used test cases generated by Evosuite [Fraser and Arcuri(2011)] because, according to the results from He et al. [Ye et al.(2021b)], Evosuite outperforms Randoop in the detection of incorrect patches. We applied the scripts from RGT that remove flaky test cases that could be generated by Evosuite.

We ran RGT using Evosuite tests on all the unlabelled 366 patches. RGT applies a patch to the buggy version and then executes the Evosuite test cases. Those patches that have at least one associated failing test case are classified as overfitting. To validate the correctness of RGT, we then sample a subset of them (10% from the total of patches analysed by RGT) and check if those patches are actually incorrect. The manual assessment that was done by two authors of the paper did not find any false positive reported by RGT.

Nevertheless, the patches that are not classified by RGT as overfitting (i.e., no test case from RGT fails on the patched version) cannot be considered at that point as correct. That is because Evosuite may not able to generate a test that exposes the incorrect behaviour of a overfitting patch. Consequently, we proceeded with a manual analysis, explained as follows. At this stage we labelled 240 patches as overfitting, leaving 222 for manual analysis.

5.2.4 Manual Assessment

For all the patches that could not be labelled by using the three methods described above (222 overall), we carry out a manual assessment of their correctness, following the recommendation by Liu et al [Liu et al.(2020)]. Each patch was independently assessed by two authors of this paper. Each assessor compared an APR-generated plausible patch for a given bug with the corresponding human-written patch (which is provided by the Defects4J framework), and labelled the plausible patch as *correct* if both patches were syntactically or semantically equivalent. Otherwise it was marked as *overfitting*. If the assessors had any disagreement on the assessment of a patch, an additional author was asked to assess the given patch. We also sampled 10% of patches for cross-checking, achieving a 100% agreement rate.

5.2.5 Final Set Of Patches

To answer our RQs, we thus report on 1,223 labelled patches, 572 of which are syntactically unique, generated by 10 tools for 45 bugs within an 8-hour time limit (Table 2). Since for RQs 1–4 we consider 3-hour time limit, we present the numbers in the table too. Overall, 155 bugs were patched, with 43 bugs fixed in the 3-hour time limit.

6 Results

In this section, we present the results of our patch analysis and answer our research questions. We chose a 3-hour time limit for RQ1–4, as this is the standard for APR tool evaluation [Liu et al.(2021)]. For RQ5 we use data for the full 8-hour runs.

Table 3: RQ2 & RQ4. Number of patches 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.

Tool	Bugs fixed	Sample sufficient with x% probability: median 80%	median 90%	median 100%	maximum 80%	maximum 90%	maximum 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

6.1 RQ1: Magnitude of Overfitting

For the 10 APR tools and 43 bugs the median number of patches generated per tool per bug is 2, while the average is 4.11, with a max of 45 (produced by the tool kPAR for the Math-50 bug).

This is already a striking result. 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.

Interestingly, for all 10 tools, the median number of unique patches generated for the 155 bugs (for which patches have been generated by at least one tool) is 2, while the average is 6.71.

Answer to RQ1: Our study reveals that the median number of patches generated per tool per bug is 2. Although APR tools generated patches for only 155 out of 395 bugs, developers might still consider using such tools, as they would likely only need to analyse two patches per bug.

6.2 RQ2: Overfitting Per Tool

Table 3 presents the number of patches a developer using a particular tool would have to randomly sample in order to have 80%, 90% or 100% chance of selecting a correct patch for a given bug, by using the RS baseline outlined in Section 2. A median value of 1 or 0 means that for most bugs the patches generated by a given tool are either 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.

For half of the tools the maximum value of patches one needs to sample to have 90% chance of finding a correct patch (if one was generated) for a given bug is 3 (see second-to-last column in Table 3). 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. Therefore, if a developer wants to use either SimFix or TBar for bugs similar to those in our dataset, one could sample just one patch at random, and in most cases, they would know if a correct patch was generated. Furthermore, for SimFix they might consider up to 3 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.[7] 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 3). In particular, we note that TBar is able to find the largest number of fixes, i.e., 25, for 43 bugs for

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

Table 4: RQ3 & RQ4. Number of patches 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 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				

which known fixes were found by any of the 10 tools. However, it requires the largest number of patches to be sampled randomly to establish whether a correct patch was generated or not, pointing out that the internal patch assessment used by TBar could be further improved.

Answer to RQ2: Our study reveals that one needs to randomly sample up to 6 unique patches per tool for most bugs to have confidence whether a correct patch was generated or not. SimFix outperforms other tools in that regard, being the second best tool in terms of efficacy (finding fixes for 19 bugs), while needing only up to 3 randomly sampled patches to review per bug.

6.3 RQ3: 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.

Answer to RQ3: Our study reveals that for 34 out of 43 bugs, one has a 80% chance of selecting a correct patch if only up to 6 patches are sampled, while for half of the bugs only 3 patches need to be evaluated. These numbers are not much higher than for an individual tool. Therefore, the use of multiple tooling to select a set of patches might be a viable approach to increase chances of finding a fix.

6.4 RQ4: Patch Sampling Strategy

In the previous two questions (RQs 2-3), we looked at two scenarios: one in which we sample plausible patches for assessment from a single tool (RQ2); and one where we gather all plausible patches generated by all given tools and sample a patch from that set (RQ3). Whereas in RQ4 we discuss in which circumstances which strategy is more beneficial.

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 3 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 3 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.

Therefore, whether it is best to consider the patches of individual tooling or combine patches together to sample from a bigger sample will depend on the choice of tools and bugs used, thus we advocate both sampling strategies are considered. Our initial empirical results provide a probabilistic estimate for each of the tools and bugs. We acknowledge that the results will differ, depending on the bugs and tools used. However, future research on post-APR overfitting that uses the Defects4J dataset should take into account the empirical baselines established here.

Answer to RQ4: Our study reveals that when using multiple tools to generate plausible patches, it is worth considering which tool the patches came from at the patch assessment stage. Prioritising patches from tools that have a higher effectiveness rate and generate fewer patches needed for assessment could significantly reduce the overall patch assessment effort.

6.5 RQ5: APR Tool Usage

To answer RQ5, we analyse how the 10 APRs tools we considered behave when executed up to 8 hours, as depicted in Figure 2 and in Table 5, providing more details on our artefact page.[8]

First of all, we observe that when the 10 tools run for a longer time, although some (e.g., TBar, FixMiner) keep finding correct patches, the rate decreases with time. Moreover, beyond 4.5 hours, the 10 tools are able to find fixes for only 2 additional bugs (Lang-6 and Lang-10, both found by TBar). Correct patches generated after 4.5 hours do not fix any bug that has not already been fixed by at least one of the tools before the 4.5 hour time limit. This provides us with an estimate, at least for the Defects4J set, of how long the tools should be run for (assuming the same environmental conditions).

Moreover, we analysed the first *n* patches (where *n*=1,2,..,37) generated per bug for all 10 tools, within the 8 hour limit. The results are summarised in Table 5. We observe that: (i) for 41 of the bugs, a correct patch was generated as the first patch by at least one of the tools; (ii) for 43 bugs a correct patch was found if just the first two patches were analysed per tool; (iii) correct patches for the remaining 2 bugs were found only when we considered the first 21 patches generated by each tool.

Answer to RQ5: Our study reveals that beyond 4.5 hours no new patches, syntactically equivalent to known fixes, were found. Furthermore, TBar, the tool that was able to find fixes for the largest number of bugs, is the tool that is able to fix more bugs, if given more time. We found that in order to find all patches that are syntactically equivalent to known fixes for most (43 out of 45) bugs, one only needs to consider the first two patches per tool when using multiple tools (i.e., either the correct patch was generated or not).

[8] `https://github.com/SOLAR-group/overfitting-baseline-artefact/tree/main/saved_results/paper_results/rq5/`

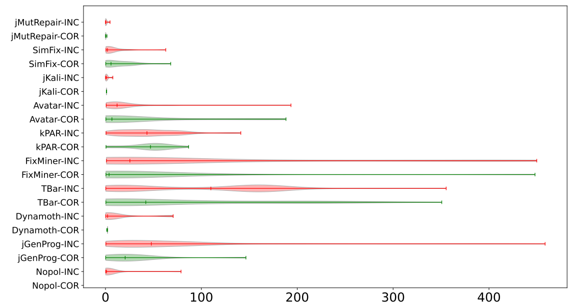

Figure 2: RQ5. Time (in min) in which each APR tool finds correct (green bar) and incorrect patches (red bar).

Table 5: RQ5. 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 first *n* 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

7 Related Work

Software engineering research has long advocated for comparing novel approaches against existing alternatives, or in their absence, against a simpler baseline [Ralph et al.(2020), Cohen(1995)]. For example, the ACM empirical standard on optimisation studies recommends that if a novel approach, for which there is no state-of-the-art available, is proposed then it should be benchmarked against random search.[9] Various studies have shown that such benchmarking is crucial to ameliorate conclusion instability in empirical SE research [Sarro and Petrozziello(2018), Chen et al.(2019), Langdon et al.(2016), Krishna and Menzies(2019)]. However, the use of a baseline benchmark for APR studies is not a common practice yet.

Moreover, although previous work has empirically assessed the effectiveness of their proposed patch overfitting assessment approaches, they all overlooked the extent to which they are beneficial in a practical scenario that reflect their usage in the real-word. In fact, the effectiveness of these approaches has been generally assessed in terms of patch reduction from a set of plausible patches. These patches were generated for a given bug in different previous work, where the APR tool was usually stopped at the first patch found or have different configuration settings. This may be due to the fact that it was convenient to reuse existing labelled patches from previous work instead of running additional experiments hence the majority of the studies focused on the evaluation of a very large number of existing patches, which exacerbated the perception of the magnitude of the manual patch assessment problem. Although such evaluations are valid in their own right, we argued that we need to consider a more practical scenario.

We are the first to investigate the magnitude of how the overfitting problem affects the practitioner, where a developer runs one or more APR tools for a given bug, within a fixed time budget, before the patch overfitting assessment takes place. Future research could use our dataset of patches generated in such a practical real-world scenario, as well as the RS baselines for that dataset, for future comparisons of patch assessment approaches.

[9] `https://acmsigsoft.github.io/EmpiricalStandards/docs/?standard=OptimizationStudies`

8 Threats to Validity

Internal Validity. The internal validity of our study may be affected by the tools we considered. However, we relied on the criteria proposed in previous work [Liu et al.(2020)] to identify a well-established set of open-source tools, which could be fit to the purpose of this study as we described in Section 5. For example, we did not include APR tools based on neural-networks evaluated on Defects4J because these tools are either *a)* not publicly available (e.g., CURE [Jiang et al.(2021)]) or *b)* require substantial modifications to produce a well-formed patch (e.g., CoCoNut [Lutellier et al.(2020)]). Starting from an initial set of 16 open-source tools, we refined it to a final set of 10 tools as we discarded three tools that are re-implementations of approaches we already consider, and other three due to likely bugs (see Section 5.1 for more detail). Regardless, the random selection baseline is general enough to be applicable to assess the impact of the overfitting problem suffered by any APR tool in practice. The fact that we do not evaluate all state-of-the-art APR tooling does not invalidate our results and the novelty of our work: Any approach that claims to reduce patch assessment due to the overfitting problem should use the RS baseline (regardless of tooling). No previous work has considered that, that is why our work is novel beyond the tools considered.

Moreover, although we are generally consistent with previous work [Liu et al.(2020)], we acknowledge some minor differences.

As shown in previous work previous labelling may contain errors [Wang et al.(2020)]. To mitigate such a threat, two of the authors who have 15+ years of Java experience, verified all those patches considered herein that were labelled as correct in previous work, and found 3 mislabelled ones which were therefore excluded from our study. [10]

Some of the APR tools we considered are heuristic-based and we run them once only in order to emulate a real-world scenario, thus it should not be surprising if results vary between runs. There is also some randomness associated with the use of test-case generation tools such as EvoSuite [Fraser and Arcuri(2011)], because of the stochastic nature of the algorithm they use. Therefore, the results of our experiments may vary between different executions. To mitigate this threat, we relied on a set of generated test cases that were generated on multiples seeds in previous work – that is, for a given buggy program (patched with the developer-written patch), He et al. [Ye et al.(2021b)] carried out 30 executions of Evosuite, each using a different random seed value, while Shamshiri et al. [Shamshiri et al.(2015)] carried out 10 executions. These multiple executions of Evosuite help produce more diverse test cases. Our automated evaluation (Section 5.2.3) uses all of those test cases in order to increase the probability of detecting overfitting patches in an automated manner. Moreover, the test cases generated by Evosuite may be flaky. Flaky test could induce our automated assessment to produce false positives (i.e., labeling correct patches as overfitting). To reduce this threat, we execute the scripts provided by He et al. [Ye et al.(2021b)] for removing flaky tests from the set of Evosuite test cases we used.

Finally, we use a timeout as a stopping criterion for the APR tools rather than setting a limit on the number of patch candidates generated. However, Liu et al [Liu et al.(2021)] found that the time-to-validate cost (i.e., the use of a timeout to stop the execution of a tool) may lead to comparison biases when the results of different studies are obtained using different execution environments. This is not the case for our experiment as we executed all the approaches on the same infrastructure.

Nevertheless, to reduce the comparison bias that might occur when comparing our results to other work, we describe in detail our experimental environment (Section 5.1.4), and provide a replication package (Section 10).

Construct Validity. The main threat to the construct validity of our study is related to the manual labelling of the patches. To ensure that patches were correctly labelled we had a second round of labelling to crosscheck the results. Patches that were not as straightforward to label were discussed among the authors until a mutual decision was made. Moreover, since we used an automated method for assessing patch correctness proposed by He et al. [Ye et al.(2021b)], we also manually analyzed 10% of such automatically assigned labels to verify that the assessment approach does not produce a false positive i.e., reporting that a correct patch is overfitting.

External Validity. In this study we investigate all bugs from one dataset, Defects4J, which may limit the external validity of our findings. This choice was dictated by the data available and the aim to include as many APR techniques as possible. In fact, the majority of APR techniques work with Defects4J, and the use of other datasets would have limited the number of techniques that we could study. The size of our dataset could mitigate such external validity issues. Future research using other bug benchmarks would further reduce this threat, thus we provide all our scripts and patches for replicability purposes (Section 10). Nevertheless, the latest version of the dataset contains the same programs as version 1.5 we use, thus our results generalise to the latest set (albeit not all programs are used as version 2 was extended with new software).

[10] `https://github.com/SOLAR-group/overfitting-baseline-artefact/tree/main/ground_truth_patches/mislabelled_in_previous_work.txt`

We also acknowledge the threat of test suite coverage. However, Defects4J was specifically developed to ensure good coverage of the given bug. The same threat concerns all APR studies using this dataset. We pose that regardless of the dataset, researchers should ensure patch assessment studies compare against a random baseline. Here we provide empirical data in case they use the same dataset to speed up such comparisons.

9 Conclusions

In this paper, we outlined a baseline, based on Random Selection (RS), for patch overfitting assessment. The main premise of this work is that every post-APR overfitting approach should beat this baseline. 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. To this end, we have empirically investigated to what extent popular APR tools for Java programs suffer from the overfitting problem.

Our results confirm that ARP tools suffer from overfitting (solving 45 out of 158 bugs for which candidate plausible patches are generated) yet this problem might be negligible as we found that a developer only need to consider a median of 2 patches sampled at random to be confident to find a fix (or be sure none exists) for the Defect4J bugs considered in our study. Therefore, we recommend to always evaluate the benefit arising from the use of any post-processing APR tooling against the RS baseline outlined herein. We found that using multiple APR tools at the same time does not make the APR overfitting problem much worse, and taking into account the type of an APR tool a patch was generated from can further mitigate the problem. Letting an APR tool run at least to the second plausible patch (most of which were found within 3h in our study) increases the chance of finding a patch that correctly fixes a given bug.

10 Data Availability

All of our scripts and labelled patches are available at: `https://github.com/SOLAR-group/overfitting-baseline-artefact/tree/main`.

11 Acknowledgements

This work was partially supported by EPSRC grant no. EP/P023991/1, ERC grant no. 741278 (EPIC) and by the Spanish RyC fellowship no. RYC2021-031523-I. For the purpose of open access, the authors have applied a Creative Commons Attribution (CC BY) license to any Accepted Manuscript version arising.

References

[Aleti and Martinez(2021)] Aldeida Aleti and Matias Martinez. 2021. E-APR: Mapping the effectiveness of automated program repair techniques. *Empirical Software Engineering* 26, 5 (2021), 99. `https://doi.org/10.1007/s10664-021-09989-x`

[Chen et al.(2019)] Jianfeng Chen, Vivek Nair, Rahul Krishna, and Tim Menzies. 2019. "Sampling" as a Baseline Optimizer for Search-Based Software Engineering. *IEEE Transactions on Software Engineering* 45, 6 (2019), 597–614. `https://doi.org/10.1109/TSE.2018.2790925`

[Chen et al.(2021)] Lingchao Chen, Yicheng Ouyang, and Lingming Zhang. 2021. Fast and Precise On-the-Fly Patch Validation for All. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1123–1134. `https://doi.org/10.1109/ICSE43902.2021.00104`

[Cohen(1995)] Paul R. Cohen. 1995. *Empirical Methods for Artificial Intelligence*. MIT Press, Cambridge, MA, USA.

[Durieux et al.(2019a)] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019a. 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`

[Durieux et al.(2019b)] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019b. Empirical Review of Java Program Repair Tools: A Large-scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In

Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 302–313. https://doi.org/10.1145/3338906.3338911

[Durieux and Monperrus(2016)] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *2016 IEEE/ACM 11th International Workshop in Automation of Software Test (AST)*. 85–91. https://doi.org/10.1109/AST.2016.021

[Fraser and Arcuri(2011)] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416—-419. https://doi.org/10.1145/2025113.2025179

[Ghanbari(2020)] 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

[Ghanbari and Zhang(2019)] 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

[Jiang et al.(2018)] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 298–309. https://doi.org/10.1145/3213846.3213871

[Jiang et al.(2021)] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[Just et al.(2014)] 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

[Kechagia et al.(2019)] 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

[Koyuncu et al.(2020)] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* (2020), 1–45.

[Krishna and Menzies(2019)] Rahul Krishna and Tim Menzies. 2019. Bellwethers: A Baseline Method for Transfer Learning. *IEEE Transactions on Software Engineering* 45, 11 (2019), 1081–1105. https://doi.org/10.1109/TSE.2018.2821670

[Langdon et al.(2016)] William B. Langdon, Javier Dolado, Federica Sarro, and Mark Harman. 2016. Exact Mean Absolute Error of Baseline Predictor, MARP0. *Information and Software Technology* 73 (2016), 16–18. https://doi.org/10.1016/j.infsof.2016.01.003

[Le et al.(2017)] 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

[Le Goues et al.(2011)] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[Lin et al.(2022)] 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

[Liu et al.(2019a)] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019a. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 102–113. https://doi.org/10.1109/ICST.2019.00020

[Liu et al.(2018)] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2018. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. *IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), 1–12.

[Liu et al.(2019b)] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019b. TBar: Revisiting Template-Based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 31—-42. https://doi.org/10.1145/3293882.3330577

[Liu et al.(2021)] 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

[Liu et al.(2020)] 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

[Lutellier et al.(2020)] 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 *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/3395363.3397369

[Martinez et al.(2017)] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22, 4 (2017), 1936–1964.

[Martinez and Monperrus(2016)] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 441–444. https://doi.org/10.1145/2931037.2948705

[Martinez and Monperrus(2018)] Matias Martinez and Martin Monperrus. 2018. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *International Symposium on Search Based Software Engineering*. Springer, 65–86.

[Noller et al.(2022)] 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

[Pacheco and Ernst(2007)] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 815—-816. https://doi.org/10.1145/1297846.1297902

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

[Ralph et al.(2020)] Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, Breno Bernard Nicolau de França, Carlo Alberto Furia, Greg Gay, Nicolas Gold, Daniel Graziotin, Pinjia He, Rashina Hoda, Natalia Juristo, Barbara Kitchenham, Valentina Lenarduzzi, Jorge Martínez, Jorge Melegati, Daniel Mendez, Tim Menzies, Jefferson Molleri, Dietmar Pfahl, Romain Robbes, Daniel Russo, Nyyti Saarimäki, Federica Sarro, Davide Taibi, Janet Siegmund, Diomidis Spinellis, Miroslaw Staron, Klaas Stol, Margaret-Anne Storey, Davide Taibi, Damian Tamburri, Marco Torchiano, Christoph Treude, Burak Turhan, Xiaofeng Wang, and Sira Vegas. 2020. Empirical Standards for Software Engineering Research. https://doi.org/10.48550/ARXIV.2010.03525

[Sarro and Petrozziello(2018)] Federica Sarro and Alessio Petrozziello. 2018. Linear programming as a baseline for software effort estimation. *ACM transactions on software engineering and methodology (TOSEM)* 27, 3 (2018), 1–28.

[Shamshiri et al.(2015)] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 201–211. `https://doi.org/10.1109/ASE.2015.86`

[Smith et al.(2015)] 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*. 532–543.

[Tian et al.(2020)] 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`

[Wang et al.(2020)] 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`

[Xiong et al.(2018)] 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`

[Xiong et al.(2017)] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 416–426. `https://doi.org/10.1109/ICSE.2017.45`

[Xuan et al.(2017)] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55. `https://doi.org/10.1109/TSE.2016.2560811`

[Yang et al.(2017)] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *In Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17), 11 pages*.

[Ye et al.(2021a)] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021a. Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering* (2021).

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

[Yuan and Banzhaf(2018)] Yuan Yuan and Wolfgang Banzhaf. 2018. Arja: Automated repair of Java programs via multi-objective genetic programming. *TSE* 46, 10 (2018), 1040–1067.