A Large-Scale Empirical Review of Patch Correctness Checking Approaches

Jun Yang*
University of Illinois
Urbana-Champaign
United States
jy70@illinois.edu

Yuehan Wang*
University of Illinois
Urbana-Champaign
United States
yuehanw2@illinois.edu

Yiling Lou†
Fudan University
China
yilinglou@fudan.edu.cn

Ming Wen†
Huazhong University
of Science and Technology
China
mwenaa@hust.edu.cn

Lingming Zhang
University of Illinois
Urbana-Champaign
United States
lingming@illinois.edu

ABSTRACT

Automated Program Repair (APR) techniques have drawn wide attention from both academia and industry. Meanwhile, one main limitation with the current state-of-the-art APR tools is that patches passing all the original tests are not necessarily the correct ones wanted by developers, i.e., the *plausible patch problem*. To date, various Patch-Correctness Checking (PCC) techniques have been proposed to address this important issue. However, they are only evaluated on very limited datasets as the APR tools used for generating such patches can only explore a small subset of the search space of possible patches, posing serious threats to external validity to existing PCC studies. In this paper, we construct an extensive PCC dataset, *PraPatch* (the largest manually labeled PCC dataset to our knowledge), to revisit all nine state-of-the-art PCC techniques. More specifically, our PCC dataset *PraPatch* includes 1,988 patches generated from the recent PraPR APR tool, which leverages highly-optimized bytecode-level patch executions and can exhaustively explore all possible plausible patches within its large predefined search space (including well-known fixing patterns from various prior APR tools). Our extensive study of representative PCC techniques on *PraPatch*has revealed various findings, including: 1) the assumption made by existing static PCC techniques that correct patches are more similar to buggy code than incorrect plausible patches no longer holds, 2) state-of-the-art learning-based techniques tend to suffer from the dataset overfitting problem, 3) while dynamic techniques overall retain their effectiveness on our new dataset, their performance drops substantially on patches with more complicated changes and 4) the very recent naturalness-based techniques can substantially outperform traditional static techniques and could be a promising direction for PCC. Based on our findings, we also provide various guidelines/suggestions for advancing PCC in the near future.

*The first two authors contributed equally to this work.
†Corresponding authors.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; *Software testing and debugging*;

KEYWORDS

Patch correctness, Program repair, Empirical assessment.

ACM Reference Format:
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 '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616331

1 INTRODUCTION

Automated Program Repair (APR) [17, 23, 47, 84] aims to fix software bugs with minimal human intervention to speed up software development. Recently various APR techniques have been proposed and extensively studied in academia [19, 28, 36, 38, 61, 83] and drawn wide attention from the industry [6, 41, 45, 57, 58]. A typical APR technique first generates various candidate patches based on different strategies, such as template-based [28, 35, 36], heuristic-based [23, 84], constraint-based [17, 47, 77] and learning-based [10, 33, 42] ones; then, it validates all the candidate patches via software testing [19, 23, 42], static analysis [65], or even formal verification [8]. To date, most APR systems leverage software testing for patch validation due to the popularity and effectiveness of testing in practice.

Although test-based patch validation has been shown to be practical , it suffers from the *test overfitting issue* – patches passing all the tests (i.e., *plausible patches*) may not always be correct for all inputs [55] because software tests can hardly cover all possible program behaviors for real-world systems. Therefore, the developers are recommended to manually inspect the plausible patches to find the final *correct* ones. However, such manual inspection can be extremely challenging and time consuming given the large number of potentially plausible patches and code complexity for real-world systems [19]. To relieve the burden, various techniques have been proposed to automate Patch-Correctness Checking (PCC) – while static techniques infer patch correctness based on patched code snippets analysis [30, 68, 70], dynamic techniques rely on runtime information to determine patch correctness [76, 80]. In recent years, researchers have also leveraged the recent advances in deep code embedding for PCC [63].

While existing PCC techniques have shown promising results, they are mostly studied on limited PCC datasets, since the APR tools used for constructing such datasets can only *explore a very small portion of the possible patch search space* and are applied on a limited number of subjects. First, the leveraged APR tools often incorporate aggressive patch pruning strategies so as to fix more bugs on their studied subjects, which makes these tools only explore a small portion of the possible patch search space and may fail to fix bugs on other subjects. For example, these APR tools have been shown to be overfitting to the studied subjects, e.g., [16] showed that the studied 11 tools fix significantly more bugs in Defects4J V1.2 than other benchmarks. In addition, the widely-adopted SimFix/CapGen tools [23, 70] can only fix 0 and 2 more bugs respectively on a newer version of Defects4J with ~200 more bugs [19]. In this way, the possible plausible patches are also missed by such APR tools, and thus absent in the resulting PCC dataset. Second, due to the efficiency issue, most studied APR tools (e.g., SimFix) terminate patch exploration as soon as they find the first plausible patch. In this way, a large number of possible plausible patches can also be missed by such techniques. For example, even including 21 APR tools, the current largest manually labeled PCC dataset [66] only has 902 patches for Defects4J V1.2 in total. Third, most of the PCC datasets are built on the popular old version Defects4J V1.2 with a limited number of subjects (*i.e.*six subjects), making it unclear whether the findings can generalize to other subjects.

In this paper, we aim to revisit nine existing state-of-the-art PCC techniques with a more extensive and real-world dataset to more faithfully evaluate their effectiveness in practice. To this end, we first construct an extensive dataset, *PraPatch* (dataset for **Pra**ctical **Patch** Correctness Checking), for PCC based on the recent PraPR repair system [19]. We choose PraPR given the following reasons: (1) *its predefined patch search space is large* since it applies popular well-known fixing templates from various prior APR work [3, 53, 60], (2) *it is the only available APR tool (to our knowledge) that can exhaustively explore the entire predefined patch search space* since it generates and validates patches based on its highly optimized on-the-fly bytecode manipulation. In this way, PraPR can generate all plausible patches within its clearly defined patch search space, which can largely avoid the dataset overfitting issue [19] as an ideal candidate for constructing PCC datasets. We apply PraPR to both the widely studied Defects4J V1.2 dataset [25] with 395 bugs and the latest Defects4J V2.0 dataset with 11 additional subject systems and 401 additional bugs. In total, our dataset *PraPatch* contains 1,988 plausible patches, including 83 correct patches and 1,905 overfitting patches after manual labeling. To our knowledge, this is the largest manually labeled dataset for PCC. Then, based on our new datasets, we perform an extensive study of prior PCC techniques, including state-of-the-art static [30, 62, 68, 70, 71], dynamic [76, 80], and learning-based [63, 81] PCC techniques. Our empirical study reveals various findings and provide various guidelines/suggestions for further advancing PCC techniques.

Overall, this paper makes the following contributions:

- **Real-world Dataset.** We create a large-scale realistic dataset, *PraPatch*, for PCC. To our knowledge, this is the first dataset based on exhaustive patch search space exploration, as well as the largest manually labeled patch dataset for PCC to date.
- **Extensive Study.** We perform an extensive study of state-of-the-art PCC techniques, including static, dynamic, and learning-based ones, on our newly constructed dataset. In total, our experiments cost 580 CPU days, including static analysis, code embedding, test generation, and test execution for all patches within our dataset.
- **Practical Impacts.** Our empirical study reveals various results the community should be aware of, as well as providing various practical guidelines for future PCC work, e.g., the dynamic PCC techniques can no longer work for patches on more complex and real-world bugs and should be largely improved.

2 BACKGROUND AND RELATED WORK

In this section, we first introduce the background on automated program repair and patch correctness checking, and then motivate our study by systematically revisiting datasets used in the literature.

Automated Program Repair. Automated Program Repair (APR) [17, 23, 28, 36, 47, 61, 83, 84] aims at fixing bugs with minimal human intervention. Researchers have proposed various APR techniques, which can be categorized according to how patches are generated: (1) *Heuristic-based APR* [32, 54, 84] leverages heuristics to explore the search space of patches; (2) *Template-based APR* [28, 35, 36] incorporates patterns summarized from historical developer patches to guide patch generation; (3) *Constraint-based APR* [17, 47, 77] leverages constraint solving to directly synthesize patches; (4) *Learning-based APR* [10, 33, 42, 72, 73, 85] utilizes learning techniques, including recent Large Language Models (LLMs) [4, 9, 43, 50, 64, 67], to generate patches. APR tools then leverage software testing [19, 23, 42], static analysis [65], or even formal verification [8] to validate the patches.

Patch Correctness Checking. Most APR techniques rely on software tests to validate patches, assuming that patches that pass all tests are correct. However, this assumption can be problematic as tests may not detect all possible bugs. To address the issue of overfitting patches, Patch Correctness Checking (PCC) [29] techniques have been proposed to differentiate overfitting patches from correct patches.

There are two application scenarios for PCC techniques, with *oracle patch information* or without. In the *oracle-based* scenario, the plausible patches that exhibit differently from the oracle patch are deemed as overfitting. Note that *oracle-based* PCC is usually used for APR experimentation, and cannot be applied to real-world bug fixing where the oracle patch information is unavailable. In contrast, *oracle-absent* PCC can identify potential overfitting patches without oracle, thus can be applied in real-world bug fixing. Therefore, we only focus on the *oracle-absent* techniques.

According to whether dynamic patch execution information is required, traditional PCC techniques can be categorized into *static* and *dynamic* techniques. In particular, static techniques leverage static code features [30, 68, 70], pre-defined anti-patterns [62] or even code naturalness [71] to predict patch correctness. Dynamic techniques leverage run-time information, such as crash or memory-safety issues used in Opad [80] and test execution traces used in Patch-Sim [76]. Recently, researchers have also started exploring advanced machine learning and deep learning techniques for PCC.

Seminal research [81] characterizes patch code with pre-defined features, while more recent work [11, 63] directly encodes patch code via embedding models (e.g., BERT [15]) and learns to predict the probability of each patches being overfitting.

Existing PCC Datasets. We revisit the existing peer-refereed literature on automated program repair and PCC based on the APR review [49] before 2022, and summarize the datasets used in their evaluation. Table 1 presents the characteristics of datasets used in existing work. Note that for the patches generated by APR tools, we check whether the early stop mechanism is enabled during APR procedure as shown in Column "Early stop". From the table, we can find that although involving various APR tools, subjects, and patches, existing PCC datasets can still be insufficient for evaluating PCC, since they can only explore a very small portion of the possible patch search space on *limited bugs* due to the following reasons.

First, existing datasets mainly include the plausible patches generated by overfitting APR tools, which incorporate aggressive pruning strategies and thus could only explore a small ratio of patch search spaces. As shown in recent work [16, 19], most existing APR tools are overfitting to the Defects4J V1.2 [25] benchmark. The main reason is that such APR tools often leverage aggressive and specific patch pruning strategies so as to fix more bugs, which could explore a small ratio of patch search spaces and may suffer from the dataset-overfitting issue [61]. For example, the recent SimFix/CapGen tools [23, 70] can only fix 0/2 more bugs on additional ∼200 more bugs [19]; also, 11 APR tools have been shown to perform substantially worse on other benchmarks than the widely-used Defects4J V1.2 [16]. In this way, the possible plausible patches that should have been generated are missed, and thus absent in the resulting PCC dataset.

Second, existing datasets only include the first one/few plausible patches generated by APR tools due to efficiency issue. As shown in the table, most APR tools (e.g., SimFix [23]) employ the *early stop* mechanism, which terminates patch execution as soon as they find the first plausible patch for the sake of efficiency. Therefore, existing PCC dataset may miss a large number of possible plausible patches that can be potentially generated by these APR tools. According to Noller *et al.* [51], developers expect APR quickly (30-60min) generate 5-10 patches each bug. But among the 16 APR tools studied in [38], the most effective one, SimFix, set a timeout of 300 minutes for each bug but only generated 68 patches on Defects4J V1.2. Therefore, APR tools for previous datasets are neither practical nor efficient. In fact, even with 21 APR tools, the existing largest manually labeled PCC dataset [66] has only 902 patches for Defects4J V1.2; meanwhile a single APR (i.e., SequenceR [10]) can only generate at most 73 patches.

Third, most PCC datasets are built on the popular Defects4J V1.2 (old version) with only six subjects, making it unclear whether the findings can generalize to other subjects (e.g., the newer version Defects4J V2.0 with 11 more subjects). As shown in the table, the majority of existing PCC datasets are generated from the subjects in the most widely used benchmark Defects4J V1.2. Therefore, it remains unknown how existing PCC techniques perform on patches from other subjects.

To address the limitations above, in this work, we construct a more extensive and real-world dataset, *PraPatch*, so as to revisit all representative PCC techniques. In particular, our dataset *PraPatch* is built on the plausible patches generated by the recent byte-code level APR tool PraPR [19]. We choose PraPR since 1) its *predefined patch search space* is large since it applies popular well-known fixing templates from various prior APR work [3, 53, 60], and 2) it is the baseline APR tool that can exhaustively explore the predefined patch search space due to its highly optimized on-the-fly bytecode manipulation. Here, the *predefined patch search space* refers to all the possible patches that an APR tool could theoretically generate for a given bug (with all its applicable fix patterns in all suspicious buggy locations). For example, the template-based APR tools PraPR and TBar could generate 3,704 and 3,375 patches (which are the predefined patch space for PraPR and TBar) with their own defined fix patterns for the bug Chart-1, respectively. An *exhaustive exploration* means that all the patches in the predefined patch search space are generated and validated by the APR tool. Typically, existing APR tools suffer from large search space and time-consuming non-trivial validation, therefore, they cannot afford the exhaustive exploration and adopt the early exit mechanism to control the exploration costs; while PraPR is the only tool that is able to perform the exhaustive exploration since its highly optimized on-the-fly bytecode manipulation significantly accelerates the patch validation. In particular, PraPR is highly efficient, because it does not require compilation (it directly generates patches on bytecode level) and it can validate a bunch of patches in one JVM. For example, for the bug Chart-1, it only takes PraPR 149s in total to generate & validate all the 3,704 patches while it would take TBar 34s for even validating one single compilable patch. In this way, PraPR can generate all plausible patches within the clearly defined patch search space, which other APR tools fail to reach. Furthermore, in addition to the most widely used Defects4J V1.2 dataset [25], we further apply PraPR to the most recent version of Defects4J dataset, i.e., Defects4J V2.0 with 11 additional subject systems and 401 additional bugs.

3 DATASET CONSTRUCTION

3.1 Subjects

Our datasets are constructed with patches generated for the subjects in Defects4J [25] due to the following reasons: 1) Defects4J contains hundreds of real bugs for real-world projects and has become the most widely studied APR benchmark in the literature, 2) most prior PCC studies were performed on Defects4J [25, 63, 66, 74, 76, 82], thus enabling a more direct/fair comparison with prior work. We include all subjects from Defects4J V1.2 (with 395 bugs from 6 projects) since it is the most widely used Defects4J version in prior APR and PCC work [25, 63, 66, 68, 70, 74, 76, 82]. Furthermore, we also study the latest Defects4J V2.0, which includes 11 additional projects with 401 additional bugs(17 projects with 796 bugs in total), to study the dataset overfitting issue of existing PCC work, i.e., whether the prior PCC experimental results can generalize to the newer Defects4J version. Please refer to our website [79] for detailed statistics of subjects.

3.2 Patch Collection

Plausible Patch Collection. For each studied buggy version from Defects4J, we first run PraPR on it to generate all possible plausible patches (i.e., the patches passing all the tests) in the bytecode level. We further decompile these bytecode-level plausible patches into source-code-level patches since existing PCC techniques work at the source-code level. We make the following efforts to ensure the decompilation process as precise as possible. First, we configure PraPR to include all the required debugging information in the resulting bytecode-level patches. Second, we leverage the state-of-the-art decompiler JD-Core [1] to decompile patched bytecode files. Third, we only locate

Table 1: Datasets used in existing PCC work

Work	Studied technique	Scale	Ratio (correct/overfitting)	Dataset source		
				Subject source	**Patch source**	**Early stop**
Tan *et al.* [62]	Anti-patterns	289	30/259	CoREBench [7], GenProg [32]	**All from APR tools** GenProg, mGenProg [62], SPR [40], mSPR [62]	4/4
Xin *et al.* [74]	DiffTGen	89	10/79	Defects4J V1.2	**All from APR tools** jGenProg, jKali, NPol and HDRepair	4/4
Le *et al.* [30]	S3	85	25/60	100 bugs from 62 subjects	**All from APR tools** S3, Enumerative [56], CVC4 [56], Angelix [48]	4/4
Yang *et al.* [80]	Opad	449	22/427	45 bugs from 7 subjects	**All from APR tools** GenProg, AE [69], Kali [55], and SPR	4/4
Xin *et al.* [68]	ssFix	153	122/31	Defects4J V1.2 *exclude Mockito*	**All from APR tools** ssFix, jGenProg, jKali [46], Nopol, HDRepair [31], ACS	5/5
Xiong *et al.* [76]	Patch-Sim	139	29/110	Defects4J V1.2 *exclude Closure, Mockito*	**All from APR tools** jGenPro [46], Nopol [78], jKali, ACS [77], HDRepair	5/5
Ye *et al.* [82]	RGT	638	257/381	Defects4J V1.2	**All from APR tools** ACS, Arja, CapGen, etc., 14 tools in total	10/14
Wen *et al.* [70]	CapGen [70]	202	28/174	Defects4J V1.2 *exclude Closure, Mockito*	**All from APR tools** CapGen	0/1
Xin *et al.* [75]	sharpFix [75]	82	56/26	Bugs.jar *127 bugs from 7 projects in Bugs.jar*	**All from APR tools** sharpFix, ssFix [68]	2/2
Tian *et al.* [63]	Embedding-based	1,000	468/532	Defects4J V1.2	**778 from APR tools** RSRepair-A [84], jKali, ACS, SimFix [23], TBar [37], etc., 17 tools in total **232 from developer patches** *(*Supplemented to balance the dataset)*	12/17
Wang *et al.* [66]	12 PCC techniques	902	248/654	Defects4J V1.2	**All from APR tools** jGenProg, DynaMoth [17], SequencR [10], etc., 21 tools in total	16/21
Lin *et al.* [34]	Cache	49,694	25,589/24,105	RepairThemAll [16], ManySStuBs4J [26]	**Overfitting patches from APR tools and correct patches from developer patches** jGenProg, jKali, Nopol, etc., 11 APR tools in total and human patches	11/11
Ye *et al.* [81]	ODS	10,302	2,003/8,299	RepairThemAll, Defects4J V2.0 developer patches	**Overfitting patches from APR tools and correct patches from developer patches** jGenProg, jKali, Nopol, etc., 11 APR tools in total and human patches	11/11
PraPatch	9 PCC techniques	1,988	83/1,905	Defects4J V2.0	**All from APR tools** PraPR	0/1

the patched line in the decompiled file (note that PraPR only changes one line on bytecode level patches), and patch the original buggy source file to obtain a potential patch at this level. Finally, we perform the sanity check to ensure the decompiled source-code-level patches indeed pass all the tests. Finally, we obtain 1,988 plausible patches.

Patch Correctness Identification. For each plausible patch, we then manually determine its correctness by comparing it to the developer patch. We follow the labeling procedure in previous work [38] and the patches that satisfy the following criteria are labeled as correct: (1) the patches are syntactically identical to the developer patches, or (2) the patches are semantically equivalent to the developer patches according to the rules summarized by existing work [38]. Due to space limits, the detailed rules are listed on our website [2].

Otherwise, the patches are labeled as overfitting. We involve three participants with 3+ years Java development experience in the manual labeling procedure: two participants first label each plausible patch individually, and a third participant is then introduced to resolve the conflicts. The agreement ratio of the first two participants is 98.99%, and there are 20 out of 1,988 patches (1.01%) that receive conflict annotations from the two participants. For these 20 conflict cases, the third participant would first re-check whether they follow the ten semantically-equivalent rules. If the case is not covered by the rules, he/she would determine the correctness of the plausible patches based on the project context as well as the reference to the issue link or bug report. In this way, among all the 1,988 new plausible patches, 83 are labelled as correct and 1,905 labelled as overfitting ones.

4 STUDY DESIGN

4.1 Research Questions

Based on our newly constructed datasets, we revisit all state-of-the-art PCC techniques via the following research questions:

- **RQ1:** How does *static* PCC perform on our datasets?
- **RQ2:** How does *learning-based* PCC perform on our datasets?
- **RQ3:** How does *dynamic* PCC perform on our datasets?

4.2 Studied Techniques

Our study selects existing state-of-the-art techniques that are designed for or can be adapted to the PCC task. Specifically, the selected techniques can be broadly categorized into three categories, including *static*, *dynamic*, and *learning-based* techniques. We only include those techniques that do not require the oracle information (i.e., the developer patches) since the practical usefulness of those techniques requiring the oracle information is compromised [66].

4.2.1 Static Techniques. Wang *et al.* have empirically investigated the effectiveness of static features extracted from three tools [66], namely ssFix [68], S3 [30] and CapGen [70], and then utilized such features to check patch correctness. We use identical experiment settings and methodology to assess patch correctness in new dataset.

S3: S3 proposed six features to measure the syntactic and semantic distance between patched code and the original buggy code [30]. Among them, *AST differencing* and *cosine similarity* are utilized to prioritize and identify correct patches. Specifically, the sum of these features is computed as the suspiciousness score. Patches with higher scores are more likely to be overfitting according to the previous study [5] which claims correct patches are more similar to the original buggy code, and thus possess fewer modifications. In addition, we exclude the other three features in this study following prior work [66]. Specifically, *model counting* and *output coverage* are excluded since they cannot be generalized to all the generated patches [66]. Besides, *Anti-patterns* utilized in S3 is excluded here since we utilize it as a stand-alone tool, following the previous study [66].

ssFix: ssFix focused on the token-based syntax representation of code to identify syntax-related code fragments to generate correct patches. Specifically, *structural token similarity* and *conceptual token similarity* are calculated to measure the similarity between the buggy code and patched code. Similar to *S3*, the sum of these features is used to rank patches. Patches with higher scores are ranked higher.

CapGen: CapGen designed three context-aware features to prioritize correct patches over overfitting ones, namely the *genealogy* *similarity*, *variable similarity*, and *dependency similarity* respectively. Although such features are not initially proposed for PCC, they can still be adapted to assess patch correctness from the view of static features. Specifically, following the existing study [66], the product of these similarity scores is used to rank and prioritize patches, and patches with higher scores are considered more likely to be correct.

For the above three static techniques, we apply the Top-N strategy to identify correct patches following prior work [66]. Specifically, the Top-N prioritized patches are labeled as correct while others as overfitting, where N is the number of correct patches in our dataset following prior studies [66]. To mitigate the effect of N, we also show the average ranking of correct patches per bug (i.e., denoted as AVR) to evaluate the ability of prioritizing correct patches (detailed in Section 4.4).

Anti-patterns: Anti-patterns provides a set of generic forbidden transformations to help obtain program patches with higher qualities with minimal effort. To apply these Anti-patterns on our own dataset, we first map the Anti-patterns to PraPR's mutation rules [19] through manual inspection. If the rules fall into a specific pattern, we deem those patches generated by the mutation rule as overfitting. Such a strategy is the same as that adopted by the existing study [66]. In this way, we observe that patches generated by PraPR mainly fall into four of the seven anti-patterns, including A1: *Anti-delete CFG Exit Node*, A2: *Anti-delete Control Statement*, A5: *Anti-delete Loop-Counter* and A6: *Anti-append Early Exit*. Therefore, we classify the patches that fall into these rules as overfitting and the others as correct.

Code naturalness: Cross entropy [14] for code measures the naturalness of code against the code language model [20]. Very recently, researchers have shown that code naturalness in terms of entropy values computed by LLMs can help rank patches for faster program repair [27, 72]. Furthermore, Xia *et al.* [71] demonstrated for the first time that it is possible to use such entropy values to perform patch correctness checking, i.e., *correct patches can be more natural than overfitting patches.* However, there is no comprehensive study on how such entropy-based techniques perform in PCC datasets to our knowledge. For a list of tokens in the corpus of LLM, the mean entropy can be calculated as the negative log probability of each token as $mean\_entropy = -\sum_{i=1}^{n} \frac{log(p_{t_i})}{n}$, where t_i refers to the ith token of the sequence and p_{t_i} refers to the model probability of token t_i. Similarly, the sum entropy can be computed as $sum\_entropy = -\sum_{i=1}^{n} log(p_{t_i})$. Patches with lower sum or mean entropies are considered more *natural* and will be ranked higher. In our evaluation, we use the model CodeT5-large [67], and follow the same experimental setting with prior work [71]. For entropy-based techniques, we only use AVR (as described in section 4.4) to evaluate the effectiveness of patch ranking since the entropy is dependent on the context of bugs and improper for comparing patches across different bugs.

4.2.2 *Dynamic Techniques.*

Dynamic techniques can capture testing behavior and result for patch assessment. Specifically, we include SOTA techniques widely studied in the literature [66, 76, 80]:

Opad: Opad is designed based on the hypothesis that *patches should not introduce new crash or memory-safety problems* [80]. Therefore, any patch violating those rules is regarded as overfitting. Note that Opad is originally designed for C programs utilizing fuzzing techniques to generate new test cases. To adapt it on Java, we leverage two state-of-the-art test generation tools, i.e., *Evosuite* [18] and *Randoop* [52], to generate test cases based on the buggy version. Opad based on *Evosuite* and *Randoop* are denoted as E-Opad and R-Opad respectively, following the previous study [66]. Specifically, for *Evosuite* and *Randoop*, we generate 30 test suites for each buggy program with a time budget of 600 seconds for each test suite utilizing existing test generation module in Defects4J framework. We finally successfully generate 44,937 test suites on 796 buggy programs in total. After test generation, we first run the generated tests (for five times) on the original buggy programs to remove flaky tests. After that, we run the remaining ones on the patched programs. If any crash occurs or any exception is thrown, Opad will identify the patch as overfitting, otherwise it will identify the patch as correct.

Patch-Sim: Patch-Sim is a similarity-based PCC tool utilizing the tests generated by automated test generation tools (Randoop in the original study [76] and both Randoop and Evosuite in Wang *et al.*'s study [66]) as the test inputs. It assumes that tests with similar executions are likely to have similar results. The basic idea of Patch-Sim is that, if the patched program behaves similarly on the passed tests and behaves differently on the failed tests compared with the origin program, the patch tends to be correct. Patch-Sim proposes to calculate the similarity of program execution behaviors (denoted as *patch distance*), and then classifies patches with it. Note that in the original study, Xiong *et al.* used Randoop to generate test suites [76]. Besides, according to Wang *et al.*'s study [66], the tests generated by Randoop perform better than those by Evosuite. We, thus, choose Randoop as the test generation tool in this study. To apply Patch-Sim on $PraPR_{v1.2}$, we generate 6,570 test cases for related projects in total by ourselves. Combining with the test suites generated by the existing study [66], we obtain 10,404 test cases for Patch-Sim.

4.2.3 *Learning-based Techniques.*

Besides the traditional PCC tools, we include novel learning-based embedding technique (hereafter denoted as *Embedding*). We also include ODS proposed by Ye *et al.* [81] which aims to learn patch correctness via feature engineering. Lin *et al.* [34] proposed Cache with similar ideas but their tools are currently not executable due to broken scripts and incomplete artifacts.

Embedding: *Code Embedding* can transform source code into distributed representations as fixed-length vectors by utilizing supervised machine learning algorithms. Tian *et al.* initially proposed utilizing code embedding techniques to identify correct patches among plausible ones [63]. To be specific, Tian *et al.* fed both the buggy and patched code fragment for each patch to embedding models (e.g., BERT [15]) to obtain embedding vectors. After generating the initial vectors, they merged the pair of vectors for each patch according to the classification algorithm [21]. With the embedding vectors, Tian *et al.* captured the crossed features (e.g., cosine similarity) required in the classification. In the end, these features were fed to the classifier for identifying if the patch is correct or overfitting. Other embedding techniques[11, 12] mostly follow highly similar ideas and cannot outperform Tian *et al.* 's work [63]. Therefore, we utilize the best-performing model in Tian *et al.* 's study, denoted as BERT-LR, as the representative in this study for further investigation. Specifically, we applied the model learned by the BERT-LR method to our dataset, to investigate how it performs. Following the experimental setting as adopted by the existing study, we utilize the dataset used in [63] for training and our collected benchmark for testing. Note that we filtered out those overlapping patches between Tian's and our datasets in the training process.

ODS: *ODS* (**O**verfitting **D**etection **S**ystem) extracts a large number of static code features and leverages ensemble learning based on decision trees to predict patch correctness [81]. For a given patch, ODS extracts 202 code features, including 150 code description features representing the characteristics of the patch's ingredients and its context at different granularity, 26 repair pattern features encoded with human knowledge on repair strategies and 26 contextual syntactic features that encode the scope and similarity information in the source code. To evaluate ODS in our dataset, we follow the experimental setup in the original paper [81]. We first use the code analyzer provided by ODS to extract features of patches in our dataset and then reuse the model trained by the authors for prediction.

4.3 Studied Datasets

Table 2: The patch datasets in this study

Subdataset ID	Subjects	APR Tools	#Patch	#Overfit	#Correct
$PraPR_{v1.2}$	Defects4J V1.2	PraPR	1,311	1,264	47
$PraPR_{v2.0}$	Defects4J V2.0	PraPR	1,988	1,905	83
$Merge_{v2.0}$	Defects4J V2.0	PraPR + 21 tools	2,760	2,489	271
$Balance_{v2.0}$	Defects4J V2.0	PraPR + 21 tools	542	271	271

4.3.1 $Wang_{v1.2}$. Wang *et al.* [66] collected patches on Defects4J V1.2 from 21 APR tools, including 16 APR tools evaluated under the same configuration by a previous study [38] and other well-known APR tools that were not included in [38], including JAID [8], SketchFix [22], CapGen [70], SOFix [39] and SequenceR [10]. They performed a plausibility check to see whether the selected patches are indeed plausible and discarded others. Finally, they obtained 902 patches in total, including 248 correct and 654 overfitting patches.

4.3.2 PraPatch. The plausible patches collected from Section 3.2 constitute our main subdataset $PraPR_{v2.0}$, with 1,988 plausible patches generated by PraPR on Defects4J V2.0. Based on the main dataset, we further construct three sub-datasets for more thorough study of PCC work. Table 2 presents the details.

$PraPR_{v1.2}$. This dataset includes all 1,311 plausible patches generated by PraPR on Defects4J V1.2, separated from the main dataset to compare with prior PCC studies on the same Defects4J version.

$Merge_{v2.0}$. Though PraPR contains overlapping patch fixing patterns with many other APR tools, it is still important to consider patches from other APR tools for more comprehensive PCC evaluation. Hence, we further combine our $PraPR_{v2.0}$ with the existing largest labeled dataset derived from APR tools, i.e., $Wang_{v1.2}$ [66] that includes 902 plausible patches generated by 21 other APR tools. After carefully removing the duplicates between $Wang_{v1.2}$ and our dataset, we finally obtain 2,760 plausible patches in this merged dataset.

$Balance_{v2.0}$. Table 2 shows that the $Merge_{v2.0}$ dataset is largely imbalanced with mostly overfitting patches. While PCC studies frequently leverage imbalanced datasets, we further construct a balanced dataset based on $Merge_{v2.0}$ for more thorough evaluations. Specifically, we keep all correct patches from $Merge_{v2.0}$ and randomly sample the same number of overfitting patches. To mitigate the bias in randomness, we repeat the random sampling process for ten times and present the average results on $Balance_{v2.0}$.

To our knowledge, the datasets newly constructed in this work are the largest manually labeled datasets for evaluating PCC. For example, PraPR alone generates 1,988 plausible patches while the existing largest PCC dataset has only 902 plausible patches (Lin *et al.* [34] and Ye *et al.* [81] have datasets with ~50,000 and ~10,000 patches but they are not manually labeled). In addition, merging the patches generated by PraPR with the existing datasets could further form a larger labeled dataset (i.e., $Merge_{v2.0}$). The large scale of our new datasets is credited to the high efficiency and exhaustive search strategies of PraPR, which help collect many plausible patches that are missed by the other APR tools due to their efficiency issues or their early termination. For example, *PraPR alone generates 39 unique plausible patches for the Defects4J bug Math-28, while the other existing 21 APR tools generate 7 plausible patches in total (5 of them are unique).* The main reason could be that other tools early stop after finding the first plausible patch while PraPR exhaustively explores the whole search space. Therefore, at least 34 out of the 39 plausible patches generated by PraPR are not visited by the existing 21 APR tools. We evaluate the nine PCC techniques on both $Wang_{v1.2}$ and $PraPatch$. Note that $Wang_{v1.2}$ is used as the baseline dataset from the previous study [66].

4.4 Metrics

Following the recent works on patch correctness checking [63, 66, 76, 80], we evaluate PCC techniques against following metrics:

- TP: # of truly overfitting patches identified as overfitting.
- TN: # of truly correct patches identified as correct.
- FP: # of truly correct patches identified as overfitting.
- FN: # of truly overfitting patches identified as correct.
- Precision: $= TP/(TP+FP)$. Denoted as $Precision$.
- Recall: $= TP/(TP+FN)$. Denoted as $Recall$.
- Recall of Correct: $= TN/(TN+FP)$. Denoted as $Recall_c$.
- PR-AUC: **A**rea **U**nder **P**recision-**R**ecall **C**urve [13].
- AVR: the **AV**erage **R**anking of correct patches for bugs which have both overfitting and correct patches.
- Correct patch ratio: $= TN/(TN+FN)$. Denoted as CPR.
- Accuracy: $= (TP+TN)/(TP+FP+TN+FN)$. Denoted as ACC.

Studies suggest that PCC techniques should not mistakenly exclude correct patches [66, 76, 81], i.e., the FP should be as low as possible. However, when the dataset is imbalanced (*i.e.*, mostly overfitting patches), $Precision$ can be biased and overrated since FP is often much smaller than TP, thus posing significant effects to the final results. Therefore, following the previous work on imbalanced datasets [76], we further include the metric $Recall_c$ (the ratio of correctly identified correct patches) for complementary fair comparisons. In fact, $Recall_c$ is crucial since rejecting a correct patch may cause a bug to be unfixed. Besides, We also utilize the PR-AUC [13] metric to eliminate the effects caused by imbalanced datasets [59].

Note that PR-AUC is not applicable for rule-based techniques including Opad, Patch-Sim, Anti-patterns and ODS since the results are discrete (either 0 or 1). For the static techniques (S3, CapGen, ssFix and code naturalness) based on feature score ranking, we leverage AVR to show their average ranking of correct patches, i.e., the smaller ranking is better. For example, given AVR *a(b)*, *a* is the average ranking of the first correct patch in each bug while *b* is the average number of patches in each bug. Note that AVR is calculated on the subset of bugs that *have both overfitting and correct patches*, since patch ranking techniques would perform the same when there are only correct or overfitting patches. In addition, the numbers of bugs with both patches/numbers of patches per bug are 50/9.1($Wang_{v1.2}$), 26/10.7($PraPR_{v1.2}$), 39/9.6($PraPR_{v2.0}$), 76/12.9($Merge_{v2.0}$), respectively. In summary, an effective PCC technique should achieve high $Precision$, $Recall$, $Recall_c$ and PR-AUC, with low AVR. For ODS evaluation we reuse their metrics with two additional ones CPR and ACC following the original study [81].

4.5 Threats to Validity

The threats to *internal* validity mainly lie in the implementation of studied PCC techniques. To reduce such threats, we reuse existing implementations whenever possible, and have carefully reviewed all our code and script. In addition, to mitigate the bias in the patch correctness labeling, we involve multiple participants, follow widely-used criteria summarized from existing work [38], and have released all our patches for public review. To mitigate the potential bias of randomness in sampling, we repeat the sampling process for ten times and use the average results. The threats to *external* validity are mainly concerned with the generalizability of our findings. We mitigate the threats from a single APR tool by further merging our datasets with previous datasets. The threats can be further reduced by including more new bugs, *e.g.*, with the *GrowingBugs* dataset [24]. The threats to *construct* validity mainly lie in the metrics used in our study. To mitigate the threats, we have included all popular metrics for PCC, including $Precision$, $Recall$, $Recall_c$, AVR, PR-AUC, CPR and ACC.

5 RESULT ANALYSIS

5.1 RQ1: Static Techniques

5.1.1 Static Code features. For studying all the static code features, we follow the same experimental setting as the recent study [66]. The detailed experimental results for the three studied techniques are shown in Table 3. From the table, we can observe that compared with original scores on $Wang_{v1.2}$, the $Recall$ scores all increase while the $Recall_c$ scores all substantially decrease on the $PraPR_{v1.2}$ and $PraPR_{v2.0}$ datasets. One direct reason is that the two new datasets are extremely imbalanced (with mostly overfitting patches). Meanwhile, the extremely low $Recall_c$ scores (i.e., only 5%-20% of correct patches are identified as correct) still demonstrate that such techniques can hardly be useful in practice. The PR-AUC scores of all features for all datasets are below or around 50%. Taking $Balance_{v2.0}$ dataset for example, two of the features get a PR-AUC score slightly over 50% (the proportion of positive sample and the expected PR-AUC for random classification model) and one of them gets less than 50%, indicating that these features are hardly useful to identify overfitting patches. The experimental results on the more balanced $Merge_{v2.0}$ and $Balance_{v2.0}$ datasets further confirm this observation. For example, on the balanced dataset $Balance_{v2.0}$, the $Recall_c$ remains similar to the prior study on $Wang_{v1.2}$ [66] (i.e., ~45%), while the $Recall$ substantially drops over 30 percentage points for all three static techniques (e.g., 78.7-46.5% for ssFix). AVR is a better metric for simulating actual efforts of selecting one correct patch from plausible patches. From the tables, we can observe that for $Wang_{v1.2}$ dataset, developers would examine 2.1~2.8 patches on average (depending on the chosen technique) until the first correct patch is found. However, when PraPR dataset is considered or merged with $Wang_{v1.2}$, the AVRs rise to 5.5~8.2. In other words, when larger patch space is included, effectiveness of all similarity-based static tools significantly degrades and the cost to identify correct patch for each bug significantly increases.

To understand the reasons behind these results, we further investigate the detailed raw patch scores computed by different tools as shown in Table 4. Note that for ssFix and CapGen, the scores reveal the **similarity** between the buggy code and the patches; while for S3, the scores represent the **edit distance** of the patches. Such techniques are designed based on the widely-accepted assumption that *correct patches should be more similar to the buggy code.* To investigate the validity of such a widely-accepted assumption, we split all the patches into four groups: $PraPR_{v1.2}$ *correct patches*, $PraPR_{v1.2}$ *overfitting patches*, $Wang_{v1.2}$ *correct patches* and $Wang_{v1.2}$ *overfitting patches*. Furthermore, we also include the *developer patches* for comparison. As shown in Table 4, if we focus on the dataset of $Wang_{v1.2}$, the average scores of correct patches for ssFix and CapGen are 10.37% and 76% higher than the overfitting patches respectively.

Besides, the average score of the correct patches for S3 is 55.56% lower than of the overfitting patches. A Mann-Whitney U Test [44] is conducted to evaluate the significance of the difference and it shows that the p-values are relatively $4.19*10^{-7}$, $3.00*10^{-11}$ and $9.72*10^{-9}$ for ssFix, CapGen and S3 respectively. That is to say, correct patches in $Wang_{v1.2}$ in general share higher similarities with the buggy code and introduce fewer modifications than overfitting patches. Such results actually support the widely-accepted assumption, and can also explain why these tools exhibit promising results on $Wang_{v1.2}$ [66] and their original publications [30, 68, 70].

However, such an assumption might no longer hold on our datasets. Specifically, on $PraPR_{v2.0}$, for ssFix and CapGen, the average similarity scores of the overfitting patches are even 2.63% and 10.81% higher than those of the correct patches (the p-values are 0.070 and 0.106 respectively). Moreover, if we compare with correct developer patches, the average score of ssFix on overfitting patches in $Wang_{v1.2}$ is also 14.66% higher than that of developer patches (with p-value of $1.01*10^{-11}$) while average scores of ssFix and CapGen on $PraPR_{v2.0}$ overfitting patches are 34.48% and 57.69% higher than that of developer patches (with p-values of $1.42*10^{-54}$ and $5.53*10^{-12}$). Such results are contradictory to those observed merely based on $Wang_{v1.2}$ [66]. For S3, despite the results observed on $PraPR_{v2.0}$ share a similar trend with those on $Wang_{v1.2}$, we can still observe contradictory results if compared with developer patches. Specifically, the average distance scores of S3 on developer patches are

Table 3: Performance of static features on different datasets

	Dataset	*TP*	*FP*	*TN*	*FN*	$Precision$	$Recall$	$Recall_c$	*PR-AUC*	*AVR*
S3	$Wang_{v1.2}$	517	137	111	137	79.1%	79.1%	44.8%	45.3%	2.8(9.1)
	$PraPR_{v1.2}$	1222	42	5	42	96.7%	96.7%	10.6%	4.1%	8.2(10.7)
	$PraPR_{v2.0}$	1826	79	4	79	95.9%	95.9%	4.8%	5.0%	7.2(9.6)
	$Merge_{v2.0}$	2250	239	32	239	90.4%	90.4%	11.8%	15.3%	7.0(12.9)
	$Balance_{v2.0}$	120	151	120	151	44.3%	44.3%	44.3%	51.1%	2.39(5.24)
ssFix	$Wang_{v1.2}$	515	139	109	139	78.7%	78.7%	44.0%	43.8%	2.7(9.1)
	$PraPR_{v1.2}$	1221	43	4	43	96.6%	96.6%	8.5%	8.7%	6.8(10.7)
	$PraPR_{v2.0}$	1826	79	4	79	95.9%	95.9%	4.8%	6.7%	6.3(9.6)
	$Merge_{v2.0}$	2242	247	24	247	90.1%	90.1%	8.9%	10.1%	6.1(12.9)
	$Balance_{v2.0}$	126	145	126	145	46.5%	46.5%	46.5%	48.3%	2.27(5.48)
CapGen	$Wang_{v1.2}$	510	144	104	144	78.0%	78.0%	41.9%	46.2%	2.1(9.1)
	$PraPR_{v1.2}$	1222	42	5	42	96.7%	96.7%	10.6%	14.9%	7.4(10.7)
	$PraPR_{v2.0}$	1827	78	5	78	95.9%	95.9%	6.0%	11.4%	6.6(9.6)
	$Merge_{v2.0}$	2253	236	35	236	90.5%	90.5%	12.9%	16.9%	5.5(12.9)
	$Balance_{v2.0}$	133	138	133	138	49.1%	49.1%	49.1%	54.8%	1.98(5.20)

Table 4: Average scores based on static features

Patches	*ssFix*	*S3*	*CapGen*
$PraPR_{v2.0}$Correct	1.52	10.60	0.37
$PraPR_{v2.0}$Overfitting	1.56	15.73	0.41
$Wang_{v1.2}$Correct	1.49	11.71	0.44
$Wang_{v1.2}$Overfitting	1.35	26.35	0.25
Developer	1.16	40.64	0.26

54.23% higher in $Wang_{v1.2}$ and 158.36% higher in $PraPR_{v2.0}$ than overfitting patches (p-values of $1.87*10^{-25}$ and $4.92*10^{-81}$). We also investigate the difference of 8 corresponding sub-features separately which show similar trend, presented on our website [79].

Finding 1: The widely-accepted assumption made by existing similarity-based static techniques that correct patches in general share higher similarities with the buggy code is no longer valid on our new dataset with exhaustive patch generation.

Through further investigation, we observe that one major issue of similarity-based static techniques is that they can produce diverse similarity scores for semantic-equivalent patches. For instance, Listing 1 shows a simple example patch.

```
- if (this.rightBlock != null) {
+ if (false) {
      RectangleConstraint c4 = new RectangleConstraint...
```

Listing 1: A PraPR patch replacing condition with *false*

Table 5: Static PCC on semantically equivalent patches

Chart-13-mutant-6	*TokenStrct*	*TokenConpt*	*ASTDist*	*ASTCosDist*
mutate condition to false	0.829	0.773	6	0.025
remove whole block	0.344	0.419	51	0.097
Chart-13-mutant-6	*VariableDist*	*VariableSimi*	*SyntaxSimi*	*SemanticSimi*
mutate condition to false	3	0.5	0.245	0.011
remove whole block	20	0.393	0.347	0.618

This patch simply replaces the conditional expression with `false`, which is semantically equivalent to removing the whole if block. The scores generated by the three static techniques are totally different, as shown in Table 5. The "mutate condition to false" row displays the scores of the original patch, *i.e.*, replace the conditional expression with `false`, while the "remove whole block" row denotes the scores of the simplified patch, *i.e.*, simply removing the whole *if* block. Among the eight static features, the similarity scores of ssFix features of original patch are significantly higher compared with the simplified patch (*i.e.*, the patch with the whole *if* block removed) 140.99% and 84.49% respectively, while the edit distance scores for S3 of the original patch have much lower scores than the simplified patch. For CapGen features, the scores are also completely different from each other. These cases are frequently observed in our dataset, *i.e.*, a patch can be simplified to its equivalent patch. There are 54 *mutating condition to false* patches among the 1,311 $PraPR_{v1.2}$ patches, in addition to other patches with similar patterns. Such results provide stronger evidence that merely considering the syntactic similarity is inadequate for PCC while more advanced semantics-based techniques should be proposed.

Finding 2: All three studied static techniques can compute totally different scores for semantically equivalent patches, indicating that future static PCC techniques should incorporate more advanced semantics analysis.

5.1.2 *Anti-patterns.*

Table 6 presents the results of Anti-patterns [62] on our four new datasets and its original results in previous work [66]. From the table, we can observe that Anti-patterns performs significantly worse on PraPR patches by misidentifying a larger proportion of correct patches as overfitting and omitting more overfitting patches. Compared to the previous dataset, Anti-patterns exhibits both lower $Recall_c$ and $Recall$ on $PraPR_{v1.2}$. Although $Precision$ has a suspected improvement, the actual cause is the heavily imbalanced datasets, i.e., the overwhelming number of overfitting patches over the correct ones. Actually, the results on the balanced dataset $Balance_{v2.0}$ further confirm the worse performance.

Table 6: Performance of Anti-patterns

Datasets	*TP*	*FP*	*TN*	*FN*	$Precision$	$Recall$	$Recall_c$
$Wang_{v1.2}$	219	37	211	435	85.55%	33.49%	85.08%
$PraPR_{v1.2}$	174	10	37	1,090	94.57%	13.77%	78.72%
$PraPR_{v2.0}$	361	28	55	1,544	92.80%	18.95%	66.27%
$Merge_{v2.0}$	559	56	215	1,930	90.89%	22.46%	79.34%
$Balance_{v2.0}$	41	56	215	230	42.27%	15.13%	79.34%

We further looked into the root cause of Anti-patterns' poor performance on our datasets with a misclassified example generated exclusively by PraPR and has not been included in any previous dataset. Listing 2 presents an FN example that Anti-patterns fails to filter. The overfitting patch makes a subtle modification by removing the method invocation `calculateBottomInset` (i.e., only one token changed). Such a modification does not fall into any existing Anti-patterns, thus cannot be excluded during PCC. Actually, the loss of efficiency derives from the original intention of Anti-patterns. Anti-patterns was designed initially to filter out overfitting patches that have major wrongness, instead of capturing subtle changes. However, most of the patches in our dataset contain subtle changes. Meanwhile, other APR tools are also prone to generate patches with subtle changes in practice. As a result, Anti-patterns is faced with the risk of insufficiency in evaluating PCC in future work.

```
- double b = calculateBottomInset(h);
+ double b = h;
  area.setRect(area.getX() + l,
  area.getY() + t, w - l - r, h - t - b);
```

Listing 2: An overfitting patch misclassified by Anti-patterns

Finding 3: Deriving from the original intention, Anti-patterns performs much worse on our datasets due to its limited capability of identifying subtle changes in patches. Considering the widespread occurrence of subtle changes in all APR-generated patches, Anti-patterns may not be a good choice for evaluating PCC in future work.

5.1.3 *Naturalness-based techniques.*

Table 7 shows that in all datasets, both Sum Entropy and Mean Entropy can outperform all three static techniques. The box plot of distribution of rankings is listed on our website. Specifically, in terms of AVR, the Sum / Mean entropy outperforms the best performed static technique by 11.21%/11.21% (on $Wang_{v1.2}$), 30.13%/31.31% (on $PraPR_{v1.2}$), 34.82%/25.40% (on $PraPR_{v2.0}$), 44.08%/36.79% (on $Merge_{v2.0}$) and 25.60%/25.79% (on $Balance_{v2.0}$). Interestingly, our results also confirm prior work [71] that Sum Entropy performs slightly better than Mean Entropy for PCC. The reason is that Sum Entropy calculates the entire sequence entropy for code naturalness computation, and considers both code naturalness and length information. To conclude, naturalness-based techniques show greater potential in PCC and this is the first, to our best knowledge, evaluation of naturalness-based techniques on PCC datasets. We appeal to the community that more attention should be drawn into this direction.

Table 7: Performance of code naturalness

AVR	ssFix	s3	CapGen	Sum Entropy	Mean Entropy
$Wang_{v1.2}$	2.72(9.1)	2.8(9.1)	2.14(9.1)	1.9(9.1)	1.9(9.1)
$PraPR_{v1.2}$	6.77(10.69)	8.15(10.69)	7.38(10.69)	4.73(10.69)	4.65(10.69)
$PraPR_{v2.0}$	6.26(9.64)	7.15(9.64)	6.62(9.64)	4.08(9.64)	4.67(9.64)
$Merge_{v2.0}$	6.12(12.87)	7(12.87)	5.49(12.87)	3.07(12.87)	3.47(12.87)
$Balance_{v2.0}$	2.27(5.48)	2.39(5.24)	1.98(5.20)	1.49(5.26)	1.54(5.45)

Finding 4: Naturalness-based techniques can substantially outperform all static code features in patch ranking, and Sum Entropy performs slightly better than Mean Entropy.

5.2 RQ2: Learning-based Techniques

Table 8: Performance of the embedding-based technique

Datasets	TP	FP	TN	FN	Precision	Recall	$Recall_c$	PR-AUC
$Tian_{v1.2}$	85	14	16	24	85.86%	77.98%	53.33%	N/A
$PraPR_{v1.2}$	822	15	27	227	98.21%	78.36%	64.29%	11.86%
$PraPR_{v2.0}$	1,220	23	53	433	97.52%	72.81%	59.21%	15.51%
$PraPR_{v2.0-v1.2}$	398	8	26	206	96.14%	65.89%	52.94%	19.69%

5.2.1 Embedding technique. Table 8 presents the results of the embedding-based technique [63] on our datasets, i.e., $PraPR_{v1.2}$ and $PraPR_{v2.0}$. Since there is a large overlap between the training set of the embedding technique and $Merge_{v2.0}$/$Balance_{v2.0}$, i.e., patches collected by Liu *et al.* [38], we do not consider $Merge_{v2.0}$ or $Balance_{v2.0}$ in this RQ. For comparison, we also present the results of the embedding-based technique in its original paper (i.e., $Tian_{v1.2}$).

We can observe that the *Precision* increases in our new datasets while the *Recall* and $Recall_c$ drop on both $PraPR_{v1.2}$ and $PraPR_{v2.0}$. The decrease of *Recall* and $Recall_c$ becomes even larger in the newly added patches, i.e., $PraPR_{v2.0-v1.2}$. Such abnormal performance is due to the heavily imbalanced dataset. Therefore, we should pay more attention to PR-AUC, the metric which is least affected by the imbalanced dataset. Obviously, the embedding-based technique tends to perform badly on PraPR patches with PR-AUC to be less than 20% on all our new datasets. Furthermore, with the datasets to be gradually more balanced through $PraPR_{v1.2}$, $PraPR_{v2.0}$ and $PraPR_{v2.0-v1.2}$ (i.e., exclusive bugs in $PraPR_{v2.0}$), the performance of *Precision*, *Recall* and $Recall_c$ keeps dropping while the PR-AUC increases, which indicates the worse performance is more credible. The results further confirms that the embedding-based technique suffers from the dataset overfitting issue: it performs worse on the patches of subjects that are different from training set. Thus, we encourage future learning-based PCC work to consider across-dataset evaluation.

We then look into the potential reasons for such a decrement. The intuition of embedding-based PCC techniques is that the cosine similarity between the embedding vectors of correct patches and buggy code should be larger than the cosine similarity between the embedding vectors of overfitting patches and buggy code. Tian *et al.* [63] show that correct/overfitting patches in their dataset perfectly follow such an assumption. However, this assumption no longer holds on the additional patches generated by PraPR. Figure 1 presents the distribution of the cosine similarity between the embedding vectors of correct/overfitting patches and buggy code on the original dataset in the embedding work [63] (i.e., $Tian_{v1.2}$) and our datasets built on Defects4J V1.2 and additional bugs in Defects4J V2.0 (i.e., $PraPR_{v1.2}$ and $PraPR_{v2.0-v1.2}$). From the figure, we observe that different from the prior work [63], correct patches and overfitting patches share very close distributions of similarity scores on our datasets, which explains why the embedding-based technique exhibits worse performance on PraPR patches. Furthermore, correct patches even tend to have lower median similarity than overfitting patches on $PraPR_{v2.0-v1.2}$, demonstrating that the assumption made by the embedding-based work no longer holds on our new dataset.

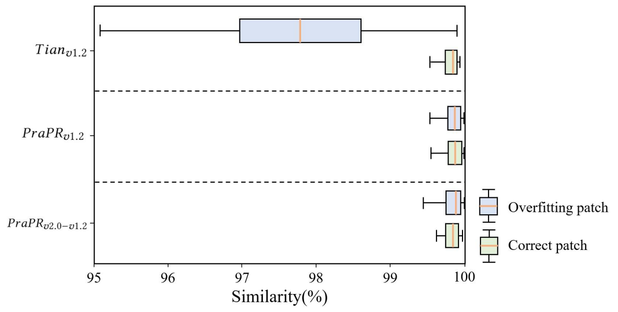

Figure 1: Similarity distribution on different datasets

Finding 5: The assumption that the cosine similarity between the embeddings of correct patches and buggy code should be larger than that between the embeddings of overfitting patches and buggy code no longer holds. Also, the embedding-based technique tends to suffer from the dataset overfitting issue.

5.2.2 ODS. Table 9 shows the result of ODS in our dataset. For $Wang_{v1.2}$, we directly reuse the results from [81]. It turns out that the performance on $PraPR_{v1.2}$ and $PraPR_{v2.0}$ significantly drop for $Recall_c$, CPR and ACC, *e.g.*, CPR drops from ~84% to ~6%, which means a lot of overfitting patches "escape" ODS and get misidentified as correct. Similarly, the $Recall_c$ drops because more correct patches are classified as overfitting.

Table 9: Performance of ODS

ODS	TP	FP	TN	FN	precision	recall	$Recall_c$	F1	CPR	accuracy
$Wang_{v1.2}$	620	66	182	34	90.38%	94.80%	73.39%	92.54%	84.26%	88.91%
$PraPR_{v1.2}$	1034	32	14	229	97.00%	81.87%	30.43%	88.79%	5.76%	80.06%
$PraPR_{v2.0}$	2571	93	35	592	96.51%	81.28%	27.34%	88.24%	5.58%	79.19%

```
- double b = calculateBottomInset(h);
+ double b = trimWidth(h);
```

Listing 3: An overfitting patch that escapes ODS

More interestingly, we evaluate ODS on developer patches and found that the FP rate (ratio of developer patches identified as overfitting) is 43.52% and 41.56% on developer patches v1.2 and v2.0. To figure out the potential reason that the performance drops, we carefully check the feature definitions in ODS and understand why PraPR patches can escape ODS. ODS defines 202 features in total (in three groups). According to the feature analysis of ODS [81], the order of importance of three groups of features is: Contextual Syntactic Features (150) > Code Description Features (26) > Repair Pattern Features (26). Note that ODS features are specific characteristics of programs like whether the patch variables are local, global, primitive, *etc.*, or whether there's code move. For single-line replacement repair tool PraPR, it normally makes simple change like replacing a variable/constant/method with another one of the same type according to the mutators defined in [19]. Therefore, most PraPR patches do not affect the majority of the ODS features at all.

Listing 3 is an example patch for bug Chart-26 showing how PraPR patches escape ODS. PraPR generates 103 patches for this bug and 23 of them are classified as correct by ODS (while actually only two of them are correct). 11 of them simply replace `calculateLeftInset` with other APIs and are all identified as correct by ODS. Such cases produce large amount of FNs and thus lead to very low CPR(~6%).

Finding 6: Features of ODS are too weak to identify overfitting patches and miss a lot of overfitting patches in our dataset. Plus, the high FP rate on developer patches is unbearable for PCC.

5.3 RQ3: Dynamic Techniques

5.3.1 Opad. Table 10 and Table 11 respectively show the results for E-Opad (Opad with EvoSuite) and R-Opad (Opad with Randoop).

Table 10: Performance of E-Opad on different datasets

E-Opad	*TP*	*FP*	*TN*	*FN*	*Precision*	*Recall*	*Recall$_c$*
$Wang_{v1.2}$	92	0	248	562	100.00%	14.07%	100.00%
$PraPR_{v1.2}$	148	0	47	1,116	100.00%	11.71%	100.00%
$PraPR_{v2.0}$	267	2	81	1,638	99.26%	14.02%	97.59%
$Merge_{v2.0}$	344	2	269	2,145	99.42%	13.82%	99.26%
$Balance_{v2.0}$	48	2	269	223	96.00%	17.71%	99.26%

Table 11: Performance of R-Opad on different datasets

R-Opad	*TP*	*FP*	*TN*	*FN*	*Precision*	*Recall*	*Recall$_c$*
$Wang_{v1.2}$	67	0	248	587	100.00%	10.24%	100.00%
$PraPR_{v1.2}$	238	12	35	1,026	95.20%	18.83%	74.47%
$PraPR_{v2.0}$	346	15	68	1,559	95.84%	18.16%	81.93%
$Merge_{v2.0}$	408	15	256	2,081	96.45%	16.39%	94.46%
$Balance_{v2.0}$	44	15	256	227	74.58%	16.24%	94.46%

We can observe that except for R-Opad on $Balance_{v2.0}$, Opad achieves a precision over 95% on all the datasets, while the achieved recall ranges from 11.71% to 18.83%. The high precision with low recall achieved by Opad is consistent with previous studies [66, 80]. Besides, the $Recall_c$ also significantly outperforms other techniques with respect to precision. Such results indicate that Opad can identify most of the correct patches and rarely classifies correct patches as overfitting ones. In other words, the performance does not degrade much in our new datasets, and Opad is more stable than static techniques. This falls into our intuition since such dynamic tools based on test generation concern more towards code semantics instead of syntactic elements. Meanwhile, similar to the findings from prior work [66], Opad achieves a rather low recall on our datasets, indicating that a substantial ratio of overfitting patches are not detected. This is still a critical issue in PCC since manually filtering out overfitting patches can be extremely costly for developers [76].

Finding 7: The performance of Opad overall tends to remain similar on our new datasets, demonstrating the robustness of such dynamic techniques. Meanwhile, consistent with the findings in prior work, it achieves a rather low recall for incorrect patches, which compromises its practical usefulness.

Though the precision achieved by Opad is still high in our new datasets, it can no longer achieve 100% preision, i.e., a few correct patches are misidentified as overfitting. This is inconsistent with the results in previous studies for PCC [66, 80]. Motivated by this, we further investigated the FP cases and made the following observations.

```
1 -  if (property == null) {
2 -     return this;
3 -  }
4    JsonFormat.Value format = findFormat0(serializers,property,handledType());...
```

Listing 4: A correct patch misclassified by Opad

For instance, Listing 4 shows a correct patch which is identified as overfitting by Opad. Specifically, this patch removes a `null` check for `property`. However, Opad is designed based on the tests generated on the original buggy programs, thus being unaware of such semantic changes. When stepping into method `findFormat0` (at line 4) from a test, `property` is null, which is unexpected to the tests generated on the original buggy program. Consequently, a `NullPointerException` was thrown and in `findFormat0`, and thus Opad mis-identified this patch as overfitting. Many other similar cases have been observed for Opad. The reason could be that some tests are generated on the buggy programs, based on some incorrect contracts or preconditions. When the semantics of a program change, those preconditions may not hold and thus fail a generated test.

Finding 8: The effectiveness of Opad might decay and it might mistakenly identify correct patches as incorrect when the semantic changes of the patches break certain conditions.

Since developer patches might even incur larger semantic changes, we are curious to see whether such cases also happen on developer patches. Unfortunately, there are no existing studies to our best knowledge. However, this is an important study since APR techniques can potentially fix more bugs and produce more developer patches in the near future. As shown in Table 12, E-Opad and R-Opad achieve $Recall_c$ of 86.06% and 79.02%, respectively, which is significantly lower than that on other datasets except for $PraPR_{v1.2}$. Specifically, Opad produces more FPs and achieves lower $Recall_c$ compared with previous study [66, 80].

Table 12: Opad/Patch-Sim on developer patches

	TP	*FP*	*TN*	*FN*	*Recall$_c$*
E-Opad	0	111	685	0	86.06%
R-Opad	0	167	629	0	79.02%
Patch-Sim	0	65	110	0	62.86%

```
@@ -483,9 +483,8 @@ public static int formatLongOctalOrBinaryBytes(
         ...
         if (length < 9) {...
+         } else {
+            formatBigIntegerBinary(value, buf, offset, length, negative);}
-         formatBigIntegerBinary(value, buf, offset, length, negative);
         buf[offset] = (byte) (negative ? 0xff : 0x80);
         return offset + length;
```

Listing 5: A developer patch misclassified by Opad

We further manually checked some FP cases in developer patches, and Listing 5 is an example. Specifically, the developer patch moved a statement into an `else` block with a condition `length >= 9`. The failing test expects an `IllegalArgumentException` triggered by invalid parameters of `formatBigIntegerBinary`. The original statement throwing that exception was not executed in the patched program. Therefore, the program stepped over that statement and when executing the next statement (line 7), the parameter `offset` is -1, triggering an `ArrayIndexOutOfBoundsException`.

Similar to the previous example, Opad is not aware of such semantic changes and misclassified that patch as overfitting. Actually, Opad could produce more FPs when the patches get more complicated, *i.e.*, different code structure or logic. This can be an important finding since eventually APR techniques will be conquering more and more complicated bugs, with more and more complicated patches.

Finding 9: Opad performs worse on developer patches, and should be applied with caution on (future) automated APR tools that can generate more complicated patches.

5.3.2 *Patch-Sim.* Table 13 presents the performance of Patch-Sim on previous datasets (i.e., $Xiong_{v1.2}$ from its original paper [76] and $Wang_{v1.2}$ from the previous study [66]) and our dataset. Note that due to an implementation issue [76], Patch-Sim does not support Defects4J V2.0 and subjects Closure and Mockito from Defects4J V1.2. Hence, following prior work [76], we only consider the patches supported by Patch-Sim from Defects4J V1.2. Overall, 493 overfitting and 10 correct patches in $PraPR_{v1.2}$ are used.

Table 13: Performance of Patch-Sim on different datasets

Datasets	*TP*	*FP*	*TN*	*FN*	*Precision*	*Recall*	$Recall_c$
$Xiong_{v1.2}$	62	0	29	48	100.00%	56.36%	100.00%
$Wang_{v1.2}$	249	51	186	392	83.00%	38.85%	78.48%
$PraPR_{v1.2}$	210	3	7	283	98.59%	42.60%	70.00%

As shown in the table 13, Patch-Sim performs much worse on our dataset compared with initial $Xiong_{v1.2}$ results. For example, Patch-Sim achieves 100% *Precision* and $Recall_c$ on its original dataset, whereas on PraPR patches, both metrics are degraded, aligning with previous findings [66]. Patch-Sim is designed based on the *mild consequence assumption* that a passing test should behave similarly on the correct patch and the buggy code. Therefore, the patch substantially changing the behavior of passing tests would be regarded as overfitting. Listing 6 presents a sample FP of Patch-Sim. In this example, the correct patch modifies the condition, leading to significant control-flow changes: on the buggy code, some passing tests can enter the `if` block and then throw the exception, whereas on the patch code these tests skip the `if` block and exception statement. Therefore, Patch-Sim considers the behaviors of these passing tests changed substantially, and further regards the patch as overfitting. However, correct patches are also likely to introduce large behavioral changes whereas overfitting patches can also induce tiny impact on passing tests.

```
-      if (fa * fb >= 0.0) {
+      if (fa * fb > 0.0D) {
         throw new ConvergenceException...}
```

Listing 6: A correct patch misclassified by Patch-Sim

Finding 10: The assumption that passing tests should behave similarly on correct patch code and buggy code can easily be broken when a larger patch space is considered, especially for the correct patches with non-trivial modifications on control flow.

Table 12 presents the performance of Patch-Sim on 175 developer patches from Defects4J V1.2 where Patch-Sim can be successfully applied. Interestingly, Patch-Sim performs even worse on developer patches with a $Recall_c$ of only 62.86%, i.e., mis-classifying 37.14% correct patches as incorrect. After manual inspection, we find that developer patches often involve sophisticated modifications and thus passing tests behave very differently from buggy code. For example, Listing 7 presents a developer patch which mis-identified as overfitting by Patch-Sim. The patch modifies multiple lines, and several `if-else` statements. Thus, it significantly affects the paths of some passing tests. In fact, it is prevalent that developer patches contain such sophisticated changes, and thus the assumption made by Patch-Sim can be easily violated on developer patches. Such results show that future PCC work should exercise complex patches since advanced APR tools can generate more complex patches in the near future.

```
- if(... && index < seqEnd - 1 && ...) {
+ if(... && index < seqEnd - 2 && ...) {...}
+    if(start == seqEnd) {
+        return 0;
+    }
     ...
-   while(input.charAt(end) != ';')
+   while(end<seqEnd && ((input.charAt(end)>='0' && input.charAt(end)<='9')
+      ||(input.charAt(end) >= 'a' && input.charAt(end) <= 'f')
+      ||(input.charAt(end) >= 'A' && input.charAt(end) <= 'F') ) )
       ...
+   boolean semiNext = (end != seqEnd) && (input.charAt(end) == ';');
-   return 2 + (end - start) + (isHex ? 1 : 0) + 1;
+   return 2 + (end - start) + (isHex ? 1 : 0) + (semiNext ? 1 : 0);
```

Listing 7: A developer patch misclassified by Patch-Sim

Finding 11: Patch-Sim misclassifies 30% correct patches as incorrect on $PraPR_{v1.2}$ and tends perform even worse with 38.14% misclassifion rate on complicated developer patches. Our results also suggest that future dynamic PCC techniques should consider more complex patches for evaluation.

6 CONCLUSION

This paper constructed a comprehensive PCC dataset, *PraPatch*, for revisiting state-of-the-art PCC techniques. Our study has revealed various interesting findings, such as 1) the similarity-based assumptions no longer hold in the new dataset, 2) naturalness-based techniques can substantially outperform static code features, 3) embedding-basedtechniques and feature engineering (ODS) can misclassify a lot of correct patches and 4) performance of dynamic techniques remain stable in the new dataset but will decay when handling patches with complex semantic changes (like developer patches). These findings may save a lot research efforts for the research community and point out future directions, like leveraging LLMs for PCC and including more realistic bugs to enhance diversity of the dataset.

DATA AVAILABILITY

Please see https://github.com/claudeyj/patch_correctness or our archived artifact [79].

ACKNOWLEDGMENTS

We thank all the reviewers for their insightful comments. We also thank He Ye from KTH, Yifeng Ding, Steven Xia and Jiawei Liu from UIUC for their help. This work was partially supported by the Natural Science Foundation under grants CCF-2131943 and CCF-2141474, the National Natural Science Foundation of China under grants No.62002125 and No.61932021, the Young Elite Scientists Sponsorship Program by CAST grant No.2021QNRC001, as well as Kwai Inc.

REFERENCES

[1] 2023. JD-Core. https://github.com/java-decompiler/jd-core.

[2] 2023. Patch manual inspection rules. https://github.com/claudeyj/patch_correctness/blob/master/semantic_equivalence_rules.md.

[3] 2023. Pitest tool. https://github.com/hcoles/pitest.

[4] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988* (2023).

[5] Moumita Asad, Kishan Kumar Ganguly, and Kazi Sakib. 2019. Impact Analysis of Syntactic and Semantic Similarities on Patch Prioritization in Automated Program Repair. In *ICSME 2019*. 328–332. https://doi.org/10.1109/ICSME.2019.00050

[6] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2021. On the Effectiveness of Unified Debugging: An Extensive Study on 16 Program Repair Systems. In *ASE 2020*. Association for Computing Machinery, New York, NY, USA, 907–918. https://doi.org/10.1145/3324884.3416566

[7] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: Studying Complexity of Regression Errors. In *ISSTA 2014*. Association for Computing Machinery, New York, NY, USA, 105–115. https://doi.org/10.1145/2610384.2628058

[8] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based program repair without the contracts. In *ASE 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 637–647. https://doi.org/10.1109/ASE.2017.8115674

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[10] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus. 2021. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *TSE* 47, 09 (sep 2021), 1943–1959. https://doi.org/10.1109/TSE.2019.2940179

[11] Viktor Csuvik, Dániel Horváth, Ferenc Horváth, and László Vidács. 2020. Utilizing Source Code Embeddings to Identify Correct Patches. In *IBF 2020*. 18–25. https://doi.org/10.1109/IBF50092.2020.9034714

[12] Viktor Csuvik, Dániel Horváth, Márk Lajkó, and László Vidács. 2021. Exploring Plausible Patches Using Source Code Embeddings in JavaScript. In *2021 IEEE/ACM International Workshop on Automated Program Repair (APR)*. 11–18. https://doi.org/10.1109/APR52552.2021.00010

[13] Jesse Davis and Mark Goadrich. 2006. The Relationship between Precision-Recall and ROC Curves *(ICML '06)*. Association for Computing Machinery, New York, NY, USA, 8 pages. https://doi.org/10.1145/1143844.1143874

[14] Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. 2005. A tutorial on the cross-entropy method. *Annals of operations research* 134 (2005), 19–67. https://doi.org/10.1007/s10479-005-5724-z

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[16] 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 *ESEC/FSE 2019* (Tallinn, Estonia). Association for Computing Machinery, New York, NY, USA, 302–313. https://doi.org/10.1145/3338906.3338911

[17] 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/10.1145/2896921.2896931

[18] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *ESEC/FSE 2011* (Szeged, Hungary). Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[19] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *ISSTA 2019* (Beijing, China). Association for Computing Machinery, New York, NY, USA, 19–30. https://doi.org/10.1145/3293882.3330559

[20] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *ICSE 2012* (Zurich, Switzerland). IEEE Press, 837–847.

[21] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed Representations of Code Changes. In *ICSE 2020* (Seoul, South Korea). Association for Computing Machinery, New York, NY, USA, 518–529. https://doi.org/10.1145/3377811.3380361

[22] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *ICSE 2018, Gothenburg, Sweden*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 12–23. https://doi.org/10.1145/3180155.3180245

[23] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *ISSTA 2018* (Amsterdam, Netherlands). Association for Computing Machinery, New York, NY, USA, 298–309. https://doi.org/10.1145/3213846.3213871

[24] Yanjie Jiang, Hui Liu, Nan Niu, Lu Zhang, and Yamin Hu. 2021. Extracting Concise Bug-Fixing Patches from Human-Written Patches in Version Control Systems. In *ICSE 2021* (Madrid, Spain). IEEE Press, 686–698. https://doi.org/10.1109/ICSE43902.2021.00069

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

[26] Rafael-Michael Karampatsis and Charles Sutton. 2020. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. In *MSR 2020* (Seoul, Republic of Korea). Association for Computing Machinery, New York, NY, USA, 573–577. https://doi.org/10.1145/3379597.3387491

[27] Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. 2022. Patch Generation with Language Models: Feasibility and Scaling Behavior. In *Deep Learning for Code Workshop*. https://openreview.net/forum?id=rHlzJh_b1-5

[28] 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 Softw. Engg.* 25, 3 (may 2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z

[29] Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On Reliability of Patch Correctness Assessment. In *ICSE 2019* (Montreal, Quebec, Canada). IEEE Press, 524–535. https://doi.org/10.1109/ICSE.2019.00064

[30] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In *ESEC/FSE 2017* (Paderborn, Germany). Association for Computing Machinery, New York, NY, USA, 593–604. https://doi.org/10.1145/3106237.3106309

[31] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *SANER 2016*, Vol. 1. 213–224. https://doi.org/10.1109/SANER.2016.76

[32] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *TSE 2012* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[33] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *ICSE 2020* (Seoul, South Korea). Association for Computing Machinery, New York, NY, USA, 602–614. https://doi.org/10.1145/3377811.3380345

[34] 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, Article 51 (may 2022), 29 pages. https://doi.org/10.1145/3505247

[35] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *ICST 2019, Xi'an, China*. IEEE, 102–113. https://doi.org/10.1109/ICST.2019.00020

[36] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *SANER 2019, Hangzhou, China*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 456–467. https://doi.org/10.1109/SANER.2019.8667970

[37] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *ISSTA 2019* (Beijing, China). Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/3293882.3330577

[38] 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 2020* (Seoul, South Korea). Association for Computing Machinery, New York, NY, USA, 615–627. https://doi.org/10.1145/3377811.3380338

[39] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 118–129. https://doi.org/10.1109/SANER.2018.8330202

[40] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *ESEC/FSE 2015* (Bergamo, Italy). Association for Computing Machinery, New York, NY, USA, 13 pages. https://doi.org/10.1145/2786805.2786811

[41] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can Automated Program Repair Refine Fault Localization? A Unified Debugging Approach. In *ISSTA 2020* (Virtual Event, USA). Association for Computing Machinery, New York, NY, USA, 75–87. https://doi.org/10.1145/3395363.3397351

[42] 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 2020, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 101–114. https://doi.org/10.1145/3395363.3397369

[43] Paula Maddigan and Teo Susnjak. 2023. Chat2VIS: Generating Data Visualisations via Natural Language using ChatGPT, Codex and GPT-3 Large Language Models. arXiv:2302.02094 [cs.HC]

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

[45] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. 2019. SapFix: Automated End-to-End Repair at Scale. In *ICSE-SEIP 2019* (Montreal, Quebec, Canada). IEEE Press, 269–278. https://doi.org/10.1109/ICSE-SEIP.2019.00039

[46] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java (Demo). In *ISSTA 2016* (Saarbrücken, Germany). Association for Computing Machinery, New York, NY, USA, 441–444. https://doi.org/10.1145/2931037.2948705

[47] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor. In *SSBSE 2018, Montpellier, France (Lecture Notes in Computer Science, Vol. 11036)*, Thelma Elita Colanzi and Phil McMinn (Eds.). Springer, 65–86. https://doi.org/10.1007/978-3-319-99241-9_3

[48] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *ICSE 2016* (Austin, Texas). Association for Computing Machinery, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[49] Martin Monperrus. 2020. The living review on automated program repair. (2020).

[50] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474 [cs.LG]

[51] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. In *ICSE 2022* (Pittsburgh, Pennsylvania). Association for Computing Machinery, New York, NY, USA, 2228–2240. https://doi.org/10.1145/3510003.3510040

[52] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *OOPSLA 2007* (Montreal, Quebec, Canada). Association for Computing Machinery, New York, NY, USA, 815–816. https://doi.org/10.1145/1297846.1297902

[53] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-Based Fault Localization. *Softw. Test. Verif. Reliab.* 25, 5–7 (aug 2015), 605–628. https://doi.org/10.1002/stvr.1509

[54] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *ICSE 2014* (Hyderabad, India). Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/2568225.2568254

[55] 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 *ISSTA 2015* (Baltimore, MD, USA). Association for Computing Machinery, New York, NY, USA, 24–36. https://doi.org/10.1145/2771783.2771791

[56] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 198–216.

[57] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: Effective Object Oriented Program Repair. In *ASE 2017* (Urbana-Champaign, IL, USA). IEEE Press, 648–659. https://doi.org/10.1109/ASE.2017.8115675

[58] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. In *ICSE 2019* (Montreal, Quebec, Canada). IEEE Press, 13–24. https://doi.org/10.1109/ICSE.2019.00020

[59] Takaya Saito and Marc Rehmsmeier. 2015. The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets. *PLOS ONE* 10, 3 (03 2015), 1–21. https://doi.org/10.1371/journal.pone.0118432

[60] David Schuler and Andreas Zeller. 2009. Javalanche: Efficient Mutation Testing for Java. In *ESEC/FSE 2009* (Amsterdam, The Netherlands). Association for Computing Machinery, New York, NY, USA, 297–298. https://doi.org/10.1145/1595696.1595750

[61] 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 *ESEC/FSE 2015* (Bergamo, Italy). Association for Computing Machinery, New York, NY, USA, 532–543. https://doi.org/10.1145/2786805.2786825

[62] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-Patterns in Search-Based Program Repair. In *FSE 2016* (Seattle, WA, USA). Association for Computing Machinery, New York, NY, USA, 727–738. https://doi.org/10.1145/2950290.2950295

[63] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. 2021. Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair. In *ASE 2020* (Virtual Event, Australia). Association for Computing Machinery, New York, NY, USA, 981–992. https://doi.org/10.1145/3324884.3416532

[64] Immanuel Trummer. 2022. CodexDB: Synthesizing Code for Query Processing from Natural Language Instructions Using GPT-3 Codex. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2921–2928. https://doi.org/10.14778/3551793.3551841

[65] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *ICSE 2018* (Gothenburg, Sweden). Association for Computing Machinery, New York, NY, USA, 151–162. https://doi.org/10.1145/3180155.3180250

[66] 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 *ASE 2020, Melbourne, Australia*. IEEE, 968–980. https://doi.org/10.1145/3324884.3416590

[67] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *CoRR* abs/2109.00859 (2021). arXiv:2109.00859 https://arxiv.org/abs/2109.00859

[68] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results. In *ASE 2013* (Silicon Valley, CA, USA). IEEE Press, 356–366. https://doi.org/10.1109/ASE.2013.6693094

[69] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 356–366. https://doi.org/10.1109/ASE.2013.6693094

[70] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *ICSE 2018* (Gothenburg, Sweden). Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233

[71] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *ICSE 2023* (Melbourne, Victoria, Australia). IEEE Press, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[72] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning. In *ESEC/FSE 2022* (Singapore, Singapore). Association for Computing Machinery, New York, NY, USA, 959–971. https://doi.org/10.1145/3540250.3549101

[73] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. arXiv:2304.00385 [cs.SE]

[74] Qi Xin and Steven P. Reiss. 2017. Identifying Test-Suite-Overfitted Patches through Test Case Generation. In *ISSTA 2017* (Santa Barbara, CA, USA). Association for Computing Machinery, New York, NY, USA, 226–236. https://doi.org/10.1145/3092703.3092718

[75] Qi Xin and Steven P. Reiss. 2019. Better Code Search and Reuse for Better Program Repair. In *GI 2019* (Montreal, Quebec, Canada). IEEE Press, 10–17. https://doi.org/10.1109/GI.2019.00012

[76] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying Patch Correctness in Test-Based Program Repair. In *ICSE 2018* (Gothenburg, Sweden). Association for Computing Machinery, New York, NY, USA, 789–799. https://doi.org/10.1145/3180155.3180182

[77] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 416–426. https://doi.org/10.1109/ICSE.2017.45

[78] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Software Eng.* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[79] Jun Yang, Yuehan Wang, Yiling Lou, Ming Wen, and Lingming Zhang. 2023. ESEC/FSE'23 Artifact for "A Large-Scale Empirical Review of Patch Correctness Checking Approaches". https://doi.org/10.5281/zenodo.8267827

[80] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. 831–841. https://doi.org/10.1145/3106237.3106274

[81] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. Automated Classification of Overfitting Patches with Statically Extracted Code Features. *CoRR* abs/1910.12057 (2019). arXiv:1910.12057 http://arxiv.org/abs/1910.12057

[82] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated Patch Assessment for Program Repair at Scale. *Empirical Softw. Engg.* 26, 2 (mar 2021), 38 pages. https://doi.org/10.1007/s10664-020-09920-w

[83] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2018. A Correlation Study between Automated Program Repair and Test-Suite Metrics. In *ICSE 2018* (Gothenburg, Sweden). Association for Computing Machinery, New York, NY, USA, 24. https://doi.org/10.1145/3180155.3182517

[84] Yuan Yuan and Wolfgang Banzhaf. 2020. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering* 46, 10 (2020), 1040–1067. https://doi.org/10.1109/TSE.2018.2874648

[85] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *ESEC/FSE 2021* (Athens, Greece). Association for Computing Machinery, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468544

Received 2023-02-02; accepted 2023-07-27