Benchmarking Automated Program Repair: An Extensive Study on Both Real-World and Artificial Bugs

Yicheng Ouyang*
University of Illinois Urbana-Champaign
Champaign, USA
youyang8@illinois.edu

Jun Yang*
University of Illinois Urbana-Champaign
Champaign, USA
jy70@illinois.edu

Lingming Zhang
University of Illinois Urbana-Champaign
Champaign, USA
lingming@illinois.edu

ABSTRACT

As bugs are inevitable and prevalent in real-world programs, many Automated Program Repair (APR) techniques have been proposed to generate patches for them. However, due to the lack of a standard for evaluating APR techniques, prior works tend to use different settings and benchmarks in evaluation, threatening the trustworthiness of the evaluation results. Additionally, they typically only adopt plausibility and genuineness as evaluation metrics, which may potentially mask some underlying issues in APR techniques.

To overcome these issues, in this paper, we conduct an extensive and multi-dimensional evaluation of nine learning-based and three traditional state-of-the-art APR techniques under the same environment and settings. We employ the widely studied Defects4J V2.0.0 benchmark and a newly constructed large-scale mutation-based benchmark named MuBench, derived from Defects4J and including 1,700 artificial bugs generated by various mutators, to uncover potential limitations in these APR techniques. We also apply multi-dimensional metrics, including compilability/plausibility/genuineness metrics, as well as SYE (SYntactic Equivalence) and TCE (Trivial Compiler Equivalence) metrics, to thoroughly analyze the 1,814,652 generated patches.

This paper presents noteworthy findings from the extensive evaluation: Firstly, Large Language Model (LLM) based APR demonstrates less susceptibility to overfitting on the Defects4J V1.2.0 dataset and fixes the most number of bugs. Secondly, the study suggests a promising future for combining traditional and learning-based APR techniques, as they exhibit complementary advantages in fixing different types of bugs. Additionally, this work highlights the necessity for further enhancing patch compilability of learning-based APR techniques, despite the presence of various existing strategies attempting to improve it. The study also reveals other guidelines for enhancing APR techniques, including the need for handling unresolvable symbol compilability issues and reducing duplicate/no-op patch generation. Finally, our study uncovers seven implementation issues in the studied techniques, with five of them confirmed and fixed by the corresponding authors.

*The first two authors contributed equally to this work.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3652140

CCS CONCEPTS

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

KEYWORDS

Program repair, Mutation testing, Empirical assessment.

ACM Reference Format:
Yicheng Ouyang, Jun Yang, and Lingming Zhang. 2024. Benchmarking Automated Program Repair: An Extensive Study on Both Real-World and Artificial Bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3652140

1 INTRODUCTION

Bugs are prevalent and inevitable in modern software systems. To help developers fix bugs more efficiently, Automated Program Repair (APR) techniques are proposed to repair bugs automatically. Recently, many APR techniques have been proposed [14, 18, 29, 55, 61] and attracted huge attention from both industry and academia. For example, SapFix [32] has been deployed in Meta's debugging pipeline, benefiting products used by millions of users.

The Generate and Validate (G&V) APR [34] represents a popular set of APR techniques that involve validating the generated candidate patches, e.g., by leveraging the program's formal specifications [28, 42]. However, as such formal specifications are usually not available, the most common way to validate the patches is to execute the whole test suite. In this context, the inputs for the APR techniques include a buggy program, the test suite, and the suspicious buggy locations. Their output consists of a set of patch candidates that could potentially fix the buggy program. After obtaining the patch candidates, test suites are executed to identify *plausible patches* (i.e., patches that pass the entire test suite), followed by manual inspection to determine the *genuine patches* (i.e., the patches that are actually correct).

To evaluate the APR techniques, many real-world bug datasets have been proposed, such as Bears [31], Bugs.jar [44], QuixBugs [22] and Defects4J [20]. As the most commonly used benchmark for APR, Defects4J consists of 800+ real bugs extracted from bug-tracking systems in its newest version V2.0.0, along with corresponding developer patches, executable test suites, and bug reports. Additionally, Defects4J provides a database abstraction layer featuring multiple useful APIs to check out buggy/fixed versions of programs, execute tests, collect coverage statistics, etc.

Although real-world bug datasets are frequently used to evaluate APR techniques, it is commonly observed that researchers,

when evaluating an APR technique, tend not to re-execute baseline techniques on the adopted benchmarks. Instead, they often reference results directly from the original publications of these baselines. Consequently, there are many issues in the evaluation of the state-of-the-art APR techniques: (1) The prevailing practice of referencing results from antecedent publications imposes significant constraints on the choice of benchmarks for comparative analyses, which can potentially compromise the scale and trustworthiness of evaluations. For example, the technique TBar [24] is exclusively assessed on Defects4J V1.2.0 in its original work. As a result, any attempt to compare a new technique with TBar, without re-executing TBar, is compelled to solely rely on evaluations conducted on Defects4J V1.2.0. However, prior research [12, 14, 59] has shown that many APR techniques are susceptible to overfitting on Defects4J V1.2.0. (2) They are usually evaluated and compared under different settings. For example, CoCoNut [29] uses a candidate size of 1000 to generate patches, while SequenceR [8] only uses 50 as the candidate size. (3) They are mainly evaluated with the metrics of plausibility and genuineness and lack comparisons and analyses in terms of other dimensions. For example, many APR techniques (e.g., [66], [11], [49]) do not perform compilability analyses on their generated patches, which may fail to fully study the potentials and limitations of corresponding techniques. In this work, we not only evaluate the compilability of the studied APR techniques, but also analyze the categories of compilation errors, and the prevalence of duplicate/no-op patches (i.e., patches that are syntactically equivalent to the original buggy code). (4) They rarely compare APR techniques on the same machine/environment.

This work presents a comprehensive evaluation of nine learning-based and three traditional APR techniques across multiple benchmarks and metrics under consistent environment and settings, aiming to address potential shortcomings in their original assessments and pinpoint avenues for enhancement. In our study, we assess these APR techniques across two distinct benchmarks: the Defects4J V2.0.0 benchmark and a mutation-based benchmark, MuBench, which comprises 1,700 synthetic bugs derived from programs in the Defects4J benchmark. We measure performance across multiple dimensions, including the total number of generated, compilable, plausible, and genuine patches, and the number of bugs with compilable, plausible, and genuine patches. To evaluate the performance of the studied APR techniques on the large MuBench benchmark, where it is too costly to inspect all plausible patches manually, we also employ the metrics of SYntactically Equivalent (SYE) patches (i.e., patches that are syntactically equivalent to the developer's patch) and Trivial Compiler Equivalent (TCE) patches (i.e., patches that are equivalent to the developer's patch after compilation) to approximate the genuineness metrics.

To glean insights across various facets, we conduct multi-dimensional analyses during the evaluation. For instance, we divide the Defects4J bugs into Defects4J V1.2.0 bugs dataset and Defects4J V2.0.0 additional bugs dataset to investigate the overfitting issue. We also analyze the correlation between compilability/plausibility/SYE/TCE metrics and genuineness metrics. Additionally, the performance of the studied APR techniques on the mutation-based MuBench benchmark across different mutators is assessed. Moreover, we analyze the compilation error categories of uncompilable patches and statistics of duplicate/no-op patches for each APR technique.

Our evaluations yield several key findings: First, traditional and Neural Machine Translation (NMT) based APR techniques tend to overfit on the widely studied Defects4J V1.2.0 dataset, while the Large Language Model (LLM) based one suffers less from the overfitting issue and can fix the most number of bugs. Second, learning-based and traditional APR techniques are better at fixing different types of bugs, demonstrating a promising future for combining learning-based and traditional APR techniques. Third, when comparing different APR techniques, the number of bugs with SYE, TCE, or plausible patches highly correlates with the number of bugs with genuine patches, establishing TCE as a cost-efficient metric for APR evaluations since it performs better than SYE and is less costly to compute than plausibility metric. Next, although some of the learning-based APR techniques adopt special strategies to improve patch compilability, their highest compilability rates still fall short of the template-based technique by approximately 20 percentage points, indicating that learning-based techniques still have room for improvement in enhancing patch compilability. Moreover, our study also reveals various guidelines for improving APR techniques, e.g., many APR techniques may generate large numbers of duplicate/no-op patches, suggesting future implementations should develop strategies to reduce such patches and improve APR efficiency and reliability. Lastly, our study leads to the detection of seven implementation issues in the studied techniques (five have been confirmed and fixed by the authors).

To sum up, our work makes the following contributions:

- **Multiple benchmarks for evaluation.** Apart from the Defects4J benchmark widely used in prior works, we also adopt a newly constructed large-scale mutation-based benchmark named MuBench, containing 1,700 bugs for evaluation.
- **Extensive evaluation.** We comprehensively evaluate nine learning-based and three traditional state-of-the-art APR techniques under uniform experimental settings on the same machine. We analyze a total of 1,814,652 generated patches to mitigate threats of biased evaluation results.
- **Multi-dimensional metrics for evaluation.** Extending beyond prior works, we incorporate various metrics of different aspects, i.e., compilability/plausibility/genuineness/SYE/TCE metrics, where SYE/TCE metrics have been understudied by existing APR research to the best of our knowledge.
- **Valuable guidelines for future APR research.** Our study analyzes the performance of these state-of-the-art APR techniques in multiple dimensions, uncovering many findings that are discussed at length. By identifying potential directions for improving these techniques, our research provides valuable guidelines for future APR research.
- **Reproducible artifact.** We have open-sourced the data, code, and details of all uncovered bugs from our study at [3].

2 BACKGROUND AND RELATED WORK

2.1 Automated Program Repair

Automated Program Repair (APR) [7, 10, 13, 15, 16, 26, 27, 33, 35, 36, 51, 58] aims to automatically generate patches to fix bugs, thereby reducing developers' debugging burden. Traditional APR

techniques can be mainly categorized into three types: (1) heuristic-based techniques, where heuristic strategies such as genetic programming [21] and random search [43, 53] are leveraged to guide the search of potentially correct patches, (2) template-based techniques, where fix patterns summarized by experts or mined from large projects are applied to buggy programs to generate candidate patches [14, 24] and (3) constraint-based techniques, where symbolic execution and constraint solving techniques are leveraged to extract semantic information for better patch generation [57]. Out of the traditional APR techniques, template-based ones have been shown to be the most effective [14], but they cannot fix bugs that are beyond the scope of their templates. To tackle this issue, NMT-based techniques [18, 29, 49, 62] are proposed, treating APR as a translation task to translate faulty code to correct code. Nevertheless, their effectiveness is heavily dependent on the quality of the bug-fix training datasets. More recently, LLM-based techniques have emerged [17, 52, 54–56, 64], showing superior performance over traditional and NMT-based techniques by leveraging pre-trained large language models.

In recent years, numerous empirical studies have been conducted to evaluate APR techniques. Liu et al. [25] systematically evaluated 16 Java APR techniques with a focus on their efficiency. Zhong et al. [65] performed an empirical study on six state-of-the-art NPR (Neural Program Repair) systems. They built a new benchmark for NPR systems and ran experiments to investigate their repairability, inclination, and generalizability. However, the prior studies [25, 65] were performed under the early-exit mechanism, i.e., terminating patch validation/generation upon discovering the first plausible patch, thereby missing a lot of potentially plausible/correct patches in the patch space. Noller et al. [37] performed a small-scale evaluation to showcase that different experimental setups can lead to different repair performances, underscoring the importance of fair comparisons under uniform experimental settings. Shariffdeen et al. [46] designed a fully agnostic repair platform integrating 20 APR tools and nine APR benchmarks across multiple target languages and application domains. However, they only included two learning-based tools, missing the majority of the recent state-of-the-art learning-based tools. Ye et al. [60] performed an empirical study of ten traditional APR techniques on the QuixBugs benchmark [22] and found 53.3% of the generated plausible patches were overfitting. Different from previous works, our study evaluates both learning-based and traditional APR techniques, not only on the commonly used Defects4J V2.0.0 benchmark but also on the mutation-based MuBench benchmark, with multi-dimensional metrics (SYE and TCE metrics have never been used for APR evaluation) to perform thorough and multi-dimensional analyses, with the aim of inspiring better APR research.

2.2 Mutation Testing and Bug Injection

Mutation testing aims to deliberately inject bugs into programs by mutating source code to measure the adequacy of the test suite. In the mutation testing context, a mutant, defined as a mutated program variant with an introduced bug, is considered "killed" if it yields different outputs from the original program during test executions. The mutation score is the metric of the effectiveness of the test suite, referring to the ratio of the killed mutants out of all generated mutants. Mutation testing techniques have been used in both academia and industry, e.g., Pitest [9] and Major [19]. Ma et al. [30] proposed to reduce the execution cost of mutation testing for object-oriented programs by using Mutant Schemata Generation (MSG) and bytecode translation. Schuler et al. [45] proposed Javalanche that ranks mutations by their impact on the behavior of program functions to enable efficient mutation testing. Brown et al. [4] proposed the wild-caught mutants technique to enhance mutation testing by generating potential faults more closely related to changes made by programmers. Zhang et al. [63] proposed predictive mutation testing, using a classification model based on a series of features related to mutants and tests to predict whether a mutant would be killed or remain alive without executing it. More recently, DeepMutation [48] was proposed to automatically learn mutants from software repositories for better mutant generation using NMT. Patra et al. [40] proposed SemSeed, a technique that can automatically seed bugs in a semantics-aware way. Zhao et al. [47] introduced LEAM, a syntax-guided mutation process leveraging neural program generation, demonstrating superior performance in mutation testing and related tasks such as test case prioritization and fault localization.

In addition to mutation testing, mutation bug injection techniques have been used in various areas including fault localization [39], fuzzing [50], and program repair [14]. Ye et al. [61] proposed to use bug injection to build a training dataset for learning-based APR. Meanwhile, there is limited prior work on leveraging mutation bug injection for extensive APR evaluation to the best of our knowledge.

Table 1: Defects4J projects utilized by MuBench

Project ID	Project Name	LoC	# Tests	# Seeds	# Mutants
Chart-1	JFreeChart	96382	2193	70	81006
Cli-1	Apcache commons-cli	1937	94	12	1118
Closure-1	Google Closure compiler	90697	7911	67	52384
Codec-1	Apache commons-codec	2584	206	11	4408
Collections-25	Apache commons-collections	26415	15393	45	11899
Compress-1	Apache commons-compress	6741	73	25	11054
Csv-1	Apache commons-csv	806	54	7	695
Gson-1	Google GSON	5418	720	29	2295
JacksonCore-1	Google Guava	15882	206	19	16982
JacksonDatabind-1	Jackson data bindings	42965	1098	56	14810
JacksonXml-1	Jackson XML extensions	4683	138	18	2209
Jsoup-1	Jsoup HTML parser	2546	139	13	1511
JxPath-1	Apache commons-jxpath	19373	308	34	19278
Lang-1	Apache commons-lang	21787	2291	31	22793
Math-1	Apache commons-math	84323	4378	61	121346
Mockito-13	Mockito framework	7289	946	48	2231
Time-1	Joda-Time	27801	4041	46	20257

Table 2: Mutators used by MuBench

Mutator	Description	Example
AOR	Arithmetic Operator Replacement	$a + b \rightarrow a - b$
COR	Conditional Expression Replacement	$a \,\|\|\, b \rightarrow a$
LOR	Bitwise Operator Replacement	$a \,\hat{}\, b \rightarrow a \mid b$
LVR	Literal Value Replacement	$1 \rightarrow -1$
ORU	Operator Replacement Unary	$-a \rightarrow \sim a$
ROR	Relational Operator Replacement	$a == b \rightarrow a >= b$
SOR	Shift Operator Replacement	$a >> b \rightarrow a << b$

3 STUDY DESIGN

3.1 Benchmark Construction

3.1.1 Defects4J Benchmark. To ensure a fair comparison among the state-of-the-art APR techniques, following prior studies [8, 54,

Table 3: State-of-the-art APR techniques evaluated in this work.

APR technique	Training source	# Training Instance	Features
Recoder [66]	Java projects between 2011 and 2018 on GitHub	82,868	syntax-guided edit decoder, placeholder generation
SelfAPR [61]	the perturbation bugs of subject projects	1,039,873	self-supervised training with error diagnostics
RewardRepair [62]	CoCoNut, Megadiff, CodRep and Bears	3,507,394	combination of syntactic training and semantic training
Tufano[2] [49]	BFP: commits between 2010 and 2017 on GitHub	46,680 + 52,364	NMT-based, code-abstraction
Sequencer [8]	original source of BFP	35,578	sequence to sequence learning
CoCoNut [29]	commits before 2010 on GitHub, projects from GitLab and Bitbucket	3,241,966	ensemble learning, CNN, context aware NMT
CURE [18]	commits before 2010 on GitHub, projects from GitLab and Bitbucket	4,040,000	NMT-based, code-aware search, BPE tokenization
Edits [11]	10,235 most-starred Java repositories on GitHub	55,000	transplanted NMT model
AlphaRepair [55]	-	-	cloze-style APR based on zero-shot large language models
TBar [24]	-	-	template-based technique with widely-used templates
SimFix [16]	-	-	using existing patches and similar code as donor code
PraPR [14]	-	-	template-based, bytecode transformation

*TBar, SimFix and PraPR are traditional APR techniques while the others are learning-based APR techniques. AlphaRepair is based on pre-trained LLMs and thus has no extra bug training dataset.

55], we use all of the 140 single-line bugs (bugs that can be fixed with a single-line modification) in the Defects4J V2.0.0 benchmark, as many techniques can only accept a single line of buggy code (e.g., SequenceR and AlphaRepair). Moreover, single-line bugs can provide clearer insights into the limitations of APR techniques since they are easier and often expected to be fixed.

3.1.2 MuBench Benchmark. Although previous works have shown that the existing APR techniques can fix a lot of bugs on real-world bug datasets (e.g., AlphaRepair [55] can fix 74 out of 395 bugs in Defects4J V1.2.0), it remains unclear whether these APR techniques can handle very simple bugs, which they ought to be able to fix for practical utility. In order to investigate this problem, we build a mutation-based benchmark named MuBench consisting of 1,700 simple artificial bugs generated by a variety of mutators.

Mutation seeds. We leverage the correct versions of programs in Defects4J V2.0.0 as the seeds and perform mutation to inject artificial bugs. As Defects4J V2.0.0 bugs are collected from 17 open-source projects, we randomly select Java source files from the fixed versions of the first non-deprecated bug IDs across all projects as the mutation seeds. After excluding files unsuitable for any mutator application, 592 Java source files were left to be mutation seeds. Note that the same Java source file can result in different mutated bugs. Table 1 shows the Defects4J fixed version of projects where we sample the seeds, as well as their project names, lines of code (LoC), the number of tests in the test suite, the number of seed Java files, and the number of mutants generated as artificial bugs.

Mutants generation. We employ the Major [19] mutation testing framework to mutate the seed programs. For each project, we run Major with all 7 mutators[1] in Table 2 enabled and execute the test suite for each mutant to check whether the mutants are killed (i.e., failing any test). In this study, we have limited our scope to include only these 7 mutators to focus on assessing the ability of studied techniques to fix relatively *simple* bugs. We leave the evaluation of more complex mutation bugs for future research.

Mutants filtering. Among the generated mutants, we use several filtering rules to filter out unsuitable mutants. Specifically, we exclude (1) mutants with multi-line code modifications (aligning with APR tools that can only handle one line of buggy code); (2) uncompilable mutants; (3) mutants that pass all tests.

Ultimately, we uniformly sample 100 mutants for each project from the remaining mutants, resulting in a total of 1,700 simple artificial bugs for the MuBench benchmark.

3.2 Subject APR Techniques

As learning-based APR techniques have been shown to have great potential, we follow a recent empirical study [65] on learning-based APR techniques to include Recoder [66], Edits [11], CoCoNut [29], Tufano [49], SequenceR [8] and CURE [18] as study subjects, but we do not include CODIT [5] due to the absence of preprocessing scripts. Additionally, we include some latest learning-based APR techniques published in top conferences, namely RewardRepair [62], SelfAPR [61], and AlphaRepair [55] (we use the CodeT5 version as recommended by the authors). For traditional APR techniques, we select SimFix [16], PraPR [14], and TBar [24] as they have demonstrated state-of-the-art performance. In total, we include nine learning-based and three traditional APR techniques, detailed information of which can be found in Table 3.

To ensure a fair comparison, we set the candidate number of learning-based techniques to 100. But for traditional APR techniques, we let them exhaustively generate patches in a 5-hour time limit because of the following reasons: (1) Compared to the patch generation speed of learning-based techniques, some of them can be slow (e.g., SimFix), thus it may take a very long time to generate 100 patches for each bug. (2) Some of the traditional APR techniques can not generate 100 patches due to their limited search space. (3) 5 hours is the time limit used by SimFix to generate patches. In addition, we conduct all experiments in the perfect fault localization setting (i.e., all faulty locations are assumed to be known to the APR techniques) following prior works [18, 54, 65]. Such a setting is adopted to eliminate the noises brought by inaccurate fault localization results [23, 25], thereby directly revealing the limitations in the repair capabilities of the studied APR techniques. For all other settings (e.g., context size, model temperature), we maintain consistency with the corresponding original works.

3.3 Patch Assessment

After obtaining the candidate patches, we leverage the on-the-fly patch validation tool UniAPR [6] to execute the test suites for each patch to reduce the cost of patch validation. Note that we

[1]The STD (Statement Deletion) mutator is excluded because it is typically hard for APR tools to restore a deleted statement, and the majority of studied tools only support replacement fixes.

[2]We follow the previous work [65] to name this technique.

follow the recent work [59] to validate all patches without the early-exit mechanism. Once UniAPR identifies all the plausible patches, we further perform patch correctness checking. For the patches generated on the Defects4J benchmark, we involve two authors with over three years of Java development experience to manually check the patch correctness: initially, they independently label each plausible patch; then they convene to resolve any disagreements until consensus is achieved. However, for the MuBench benchmark, given the large number of plausible patches, we use the TCE/SYE metrics to approximate the genuineness metrics, as detailed in Section 3.4.

3.4 Metrics

Our evaluation of APR techniques extends beyond the conventional plausibility and genuineness metrics to include the metrics of total patches and compilable patches, enabling more comprehensive analyses. Notably, we have also employed the SYE and TCE metrics to approximate the genuineness metrics, mitigating the high costs of patch validation and manual patch correctness checking in our large-scale APR evaluation.

In evaluating APR techniques, many APR works [55, 61, 62] tend to directly reuse the results presented in the original works of comparison baselines, rather than re-executing the baselines themselves. Consequently, comparisons are often made under different settings, e.g., numbers of candidate patches, posing a threat to the fairness of evaluation. Although often neglected, the number of generated patches and the compilable patches are two very important metrics in APR evaluation, as the former ensures evaluation fairness and the latter reflects the robustness and efficacy of the techniques.

Furthermore, the SYE patches refer to the patches that are syntactically equivalent to the developer's patch, i.e., the patched program has the same token list as the fixed program after tokenization. On the other hand, the TCE patches refer to the patches that are trivial compiler equivalent [38] to the developer's patch, i.e., making the patched program have the identical bytecode instructions as the fixed program after compilation. Based on the definitions, it is clear that TCE and SYE patches are all semantically equivalent to the developer's patches, i.e., they are all genuine patches, but not all genuine patches are TCE or SYE patches. Additionally, because syntactically different source files can be compiled into equivalent bytecode files, and syntactically equivalent source files must be compiled into the equivalent bytecode files, the set of SYE patches is a proper subset of the set of TCE patches. In summary, the relationships between adopted metrics can be depicted in Figure 1.

To identify SYE patches, we use a Java tokenizer generated by ANTLR [1] to compare the tokenization results of the compiled patched programs and the compiled human-fixed programs. To identify TCE patches, we use the bytecode analysis tool ASM [2] to compare the meta-information (e.g., class members, method signatures) and instructions of the bytecode files of the patched and the human-fixed programs after compilation. We deliberately exclude the debugging information from the bytecode files to perform a more accurate comparison. Additionally, for the 648 TCE and 500 SYE patches identified by the aforementioned approach among the patches generated for the Defects4J bugs, we manually checked them against the developer's patches and found no false positives.

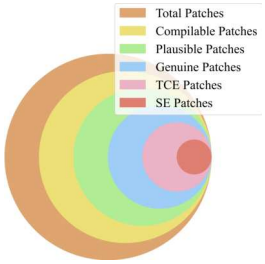

Figure 1: Metrics relationship illustration

In summary, our study primarily adopts the following metrics:

- **Compilability Metric:** A patch is compilable if the patched program can be compiled successfully.
- **Plausibility Metric:** A patch is plausible if it can successfully pass the test suite.
- **Genuineness Metric:** A patch is genuine if it is both plausible and semantically equivalent to the developer's patch.
- **SYE Metric:** A patch is SYE (SYntactically Equivalent) if, after tokenization, it shares the same list of tokens as the developer's patch.
- **TCE Metric:** A patch is TCE (Trivial Compiler Equivalent) if it has equivalent compiled bytecode compared with the developer's patch.

3.5 Research Questions

(1) **RQ1. How do the studied APR techniques perform on the Defects4J V2.0.0 single-line bugs?** This RQ seeks to evaluate the efficacy of the studied techniques on single-line bugs in the Defects4J V2.0.0 benchmark under unified settings, employing multi-dimensional metrics to uncover potential shortcomings and strengths inherent to the studied APR techniques. Additionally, we compare the results on different Defects4J versions and examine the correlation between various metrics and the genuineness metrics to enrich our insights.

(2) **RQ2. How do the studied APR techniques perform on the mutation-based MuBench benchmark?** Similar to RQ1, RQ2 aims to evaluate the performance of the studied techniques on the MuBench benchmark consisting of 1,700 simple mutation bugs. We also investigate the performance of the studied APR techniques in terms of different mutators to uncover their potential shortcomings and strengths.

(3) **RQ3. What insights can lead to better program repair?** This RQ aims to discuss additional insights into the performance of the studied APR techniques emerging from our multifaceted analyses of the evaluation, which enable us to identify potential areas for improvement and provide insightful guidance to enhance APR techniques.

4 RESULT AND ANALYSIS

4.1 RQ1: Performance on Defects4J V2.0.0 bugs

4.1.1 Overall Results. The overall results of the performance of each APR technique are shown in Table 4. Note that PraPR does not have data for compilability and SYE metrics because it generates patches at the bytecode level. Besides, not all tools can generate 100 patches for each bug. For traditional tools they cannot generate 100 patches per bug due to limited search space or low efficiency. For learning-based tools, they may fail to generate 100 patches for some bugs for intrinsic limitations, e.g., Recoder could crash on some bugs (see Section 4.3.3), and some of them would filter out invalid patches in the post-processing phase (e.g., SequenceR).

Table 4: The performance of APR techniques on Defects4J single-line bugs

Metrics	Learning-based Techniques									Traditional Techniques		
	Recoder	SelfAPR	AlphaRepair	RewardRepair	SequenceR	CoCoNut	CURE	Edits	Tufano	Tbar	SimFix	PraPR
# Total Patches	8768	14000	13439	14000	10181	12900	13900	13300	13300	10529	11499	1577
# Compilable Patches	2795 (31.88%)	5669 (40.49%)	3730 (27.76%)	4939 (35.28%)	2561 (25.15%)	2882 (22.34%)	3652 (26.27%)	291 (2.19%)	1427 (10.73%)	**6439 (61.15%)**	2699 (23.47%)	N/A
# Plausible Patches	172 (1.96%)	**658** (4.70%)	584 (4.35%)	600 (4.29%)	270 (2.65%)	245 (1.90%)	311 (2.24%)	22 (0.17%)	46 (0.35%)	364 (3.46%)	318 (2.77%)	123 (**7.80%**)
# Bugs w/ Compilable Patches	125 (89.29%)	135 (96.43%)	128 (91.43%)	**136 (97.14%)**	106 (75.71%)	117 (83.57%)	126 (90.00%)	47 (33.57%)	76 (54.29%)	128 (91.43%)	89 (63.57%)	N/A
# Bugs w/ Plausible Patches	56 (40.00%)	67 (47.86%)	**73 (52.14%)**	70 (50.00%)	44 (31.43%)	31 (22.14%)	52 (37.14%)	9 (6.43%)	17 (12.14%)	65 (46.43%)	26 (18.57%)	54 (38.57%)
# Genuine Patches	47 (0.54%)	**263 (1.88%)**	152 (1.13%)	162 (1.16%)	55 (0.54%)	65 (0.50%)	55 (0.40%)	7 (0.05%)	6 (0.05%)	52 (0.49%)	18 (0.16%)	37 (2.35%)
# TCE Patches	39 (0.44%)	**245 (1.75%)**	72 (0.54%)	116 (0.83%)	33 (0.32%)	22 (0.17%)	35 (0.25%)	7 (0.05%)	5 (0.04%)	37 (0.35%)	17 (0.15%)	20 (1.27%)
# SYE Patches	19 (0.22%)	**231 (1.65%)**	45 (0.33%)	85 (0.61%)	17 (0.17%)	19 (0.15%)	26 (0.19%)	5 (0.04%)	3 (0.02%)	34 (0.32%)	16 (0.14%)	N/A
# Bugs w/ Genuine Patches	40 (28.57%)	40 (28.57%)	**50 (35.71%)**	45 (32.14%)	27 (19.29%)	17 (12.14%)	31 (22.14%)	6 (4.29%)	6 (4.29%)	40 (28.57%)	17 (12.14%)	37 (26.43%)
# Bugs w/ TCE Patches	38 (27.14%)	35 (25.00%)	**48 (34.29%)**	44 (31.43%)	26 (18.57%)	15 (10.71%)	27 (19.29%)	6 (4.29%)	5 (3.57%)	36 (25.71%)	17 (12.14%)	20 (14.29%)
# Bugs w/ SYE Patches	19 (13.57%)	33 (23.57%)	**45 (32.14%)**	41 (29.29%)	17 (12.14%)	15 (10.71%)	26 (18.57%)	5 (3.57%)	3 (2.14%)	34 (24.29%)	16 (11.43%)	N/A

*The percentages in the parentheses denote the ratio of the number of corresponding patches to the total number of generated patches for the **# Compilable/Plausible/Genuine/TCE/SYE Patches** metrics and the ratio of the number of bugs with corresponding patches to the total number of bugs, for **# Bugs w/ * Patches**. The largest/highest number/ratio in each row is highlighted in **bold** font.

Table 5: The performance of APR techniques on different versions of Defects4J bugs

Metrics		Learning-based Techniques									Traditional Techniques		
		Recoder	SelfAPR	AlphaRepair	RewardRepair	SequenceR	CoCoNut	CURE	Edits	Tufano	Tbar	SimFix	PraPR
# Genuine Patches	**V1.2.0**	36 (0.75%)	210 (2.76%)	61 (0.84%)	125 (1.64%)	33 (0.59%)	45 (0.63%)	32 (0.43%)	2 (0.03%)	2 (0.03%)	41 (0.87%)	14 (0.19%)	28 (3.14%)
	V2.0.0	11 (0.28%)	53 (0.83%)	91 (1.48%)	37 (0.58%)	22 (0.48%)	20 (0.35%)	23 (0.36%)	5 (0.08%)	4 (0.07%)	11 (0.19%)	4 (0.14%)	9 (1.31%)
	Ratio Change	**-63.36%**	**-70.03%**	+77.18%	**-64.85%**	**-17.59%**	**-43.86%**	**-15.77%**	+213.56%	+150.85%	**-77.94%**	**-27.64%**	**-58.25%**
# Bugs w/ Genuine Patches	**V1.2.0**	32 (42.11%)	31 (40.79%)	29 (38.16%)	30 (39.47%)	18 (23.68%)	10 (13.16%)	21 (27.63%)	2 (2.63%)	2 (2.63%)	32 (42.11%)	14 (18.42%)	28 (36.84%)
	V2.0.0	8 (12.50%)	9 (14.06%)	21 (32.81%)	15 (23.44%)	9 (14.06%)	7 (10.94%)	10 (15.63%)	4 (6.25%)	4 (6.25%)	8 (12.50%)	2 (3.13%)	9 (14.06%)
	Ratio Change	**-70.31%**	**-65.52%**	**-14.01%**	**-40.63%**	**-40.63%**	**-16.88%**	**-43.45%**	+137.50%	+137.50%	**-70.31%**	**-83.04%**	**-61.83%**

*The percentages in parentheses for **# Genuine Patches** represent the ratio of the number of patches to the total number of generated patches. The percentages in parentheses for **# Bugs w/ Genuine Patches** represent the ratio of the number of bugs to the total number of bugs. Rows of **Ratio Change** indicate the percentages of the ratio decrease, with negative percentages highlighted in **bold** font.

Table 4 demonstrates that TBar achieves the highest number/ratio (6439/61.15%) of compilable patches, which is 20.66 percentage points (pp) higher than the second-best (SelfAPR, 5669/40.49%). However, despite the high compilability rate, TBar fixes fewer bugs than AlphaRepair and RewardRepair, suggesting that their effectiveness can compensate for the shortcomings in compilability. The results indicate that while some learning-based APR techniques have demonstrated better effectiveness, they still fall short of template-based APR techniques in producing compilable patches.

Moreover, SelfAPR performs the best in terms of the total number of plausible/genuine/TCE/SYE patches (658/263/245/231). However, such high numbers of patches do not necessarily lead to the highest number of bugs being correctly fixed, which is the key metric for practical applications of APR techniques. Although AlphaRepair has fewer plausible/genuine/TCE/SYE patches than SelfAPR, it attains the highest number of bugs with plausible/genuine/TCE/SYE patches, thus being the most effective APR tool among all studied APR techniques. After further looking into the patches generated by SelfAPR, we find that its good performance in the number of plausible/genuine/TCE/SYE patches is mainly caused by the high duplication rate of its generated patches (discussed in Section 4.3.2).

It is important to note that, while many of the learning-based APR techniques claim superiority over traditional APR techniques, our uniform experimental settings reveal a different narrative. Among the learning-based APR techniques, only AlphaRepair and RewardRepair can surpass TBar in terms of the number of bugs with genuine patches, contradicting the findings reported in many of their respective works (e.g., [66], [61] and [18]). Some learning-based APR techniques, including RewardRepair, SelfAPR, and CURE, directly compare their results with TBar's results in the work of TBar, neglecting the fact that TBar's assessment utilized the early-exit mechanism [24] distinct from their own. Such a mechanism, which terminates patch generation and validation once a plausible patch is identified for a bug, can lead to a potential undercount of plausible and genuine patches. We have deactivated this mechanism in our study to ensure a fair evaluation.

The contrasting results underline the importance of uniform experimental settings (e.g., patch candidate numbers and patch generation mechanisms) in evaluating APR techniques. We recommend researchers either execute APR techniques with the same experimental settings as prior works or rerun baseline APR techniques under uniform settings to ensure equitable evaluations.

> **Finding 1:** On the Defects4J benchmark, the LLM-based APR can correctly fix the most bugs while none of the learning-based tools can outperform traditional template-based TBar regarding compilability rate. In uniform settings, some NMT-based tools can not outperform TBar in bug fixing, highlighting the importance of equitable APR evaluations.

4.1.2 Performance in Different Defects4J Versions. As some APR techniques have only been evaluated on Defects4J V1.2.0, there are concerns about overfitting (i.e., the results may fail to generalize effectively to other datasets). It would be interesting to see how the results change on different benchmark versions[12, 14]. Specifically, we have divided the bugs in Defects4J into two datasets. The first with 76 bugs from Defects4J V1.2.0 projects, and the second with 64 additional bugs from Defects4J V2.0.0 projects, referred to as the Defects4J V2.0.0 dataset for brevity. The performance of studied APR techniques on these two datasets in terms of the genuineness metrics are shown in Table 5.

According to the results of the ratio change for both metrics in Table 5, most of the studied APR techniques exhibit a significant decrease in performance. Notably, 9 out of 12 APR techniques exhibit a drop in the ratio of both metrics. Among them, Recoder, SelfAPR, TBar, and PraPR experience a decrease of more than 50% in both ratios, indicating potential overfitting issues. Although Edits and Tufano exhibit increases in both ratios, the observed changes are not statistically significant due to their relatively small absolute numbers. It is worth noting that, AlphaRepair shows a 77.18% increase in the ratio of genuine patches and only a 14.01%

decrease in the ratio of correctly fixed bugs. Such results not only reaffirm the overfitting issues on Defects4J V1.2.0 identified in prior works [12, 14, 59] but also underscore the adaptability and generalization ability of the LLM-based APR, especially when confronted with a new benchmark featuring a diverse range of bugs.

Finding 2: All of the traditional techniques and six out of nine learning-based techniques exhibit overfitting issues on the Defects4J V1.2.0 dataset. Notably, the LLM-based AlphaRepair demonstrates better adaptability and generalizability concerning overfitting issues.

4.1.3 Correlation between Plausible/TCE/SYE and Genuine Patches. As genuineness metrics are usually expensive to obtain (requiring patch validation and manual patch correctness checking), we further investigate whether other metrics can serve as alternatives to them. Specifically, we perform the Pearson Correlation Coefficient analysis [41] on plausible/TCE/SYE metrics and genuineness metrics, where the results are shown in Figures 2, 3, and 4. The diagram on the left of each figure shows the correlation between the ratio of plausible/TCE/SYE patches and the ratio of genuine patches, and the diagram on the right shows the correlation between the ratio of bugs with plausible/TCE/SYE and genuine patches.

According to Figures 2, 3, and 4, the metrics of TCE/SYE/plausibility are all highly correlated to the genuineness metrics and the correlations are statistically significant at the significance level of 0.05, which implies that all of TCE/SYE/plausible metrics can potentially become good alternatives to genuineness metrics for APR evaluation. By comparing Figure 3 and Figure 4, it is obvious that the TCE metrics have a stronger correlation than the SYE metrics. Moreover, we can observe that the TCE metrics and the plausibility metrics have a similarly strong correlation with the genuineness metrics. However, after calculating the ratio of the total number of bugs with plausible/TCE/SYE patches to the total number of bugs with genuine patches, we find that the ratio for TCE (89.40%) is closer to 100% than those for plausibility (156.76%) and SYE (80.45%), which implies that the TCE metrics are the most cost-efficient alternative among other metrics to approximate genuineness metrics, given their high accuracy and efficiency (does not require test execution or manual patch correctness checking).

Finding 3: The SYE/TCE/plausibility metrics are all highly correlated to the genuineness metrics. Among them, TCE metrics serve as the most cost-efficient alternative for genuineness metrics, due to their higher accuracy than both SYE and plausibility metrics, and lower computational cost than plausibility metrics.

4.2 RQ2: Performance on the MuBench benchmark

4.2.1 Overall Results. The overall performance of APR techniques is shown in Table 6, following the same format as Table 4. Note that due to the large number of plausible patches, it is impractical to manually inspect them. Instead, we use the TCE metrics to approximate the genuineness metrics as demonstrated in Section 4.1.3.

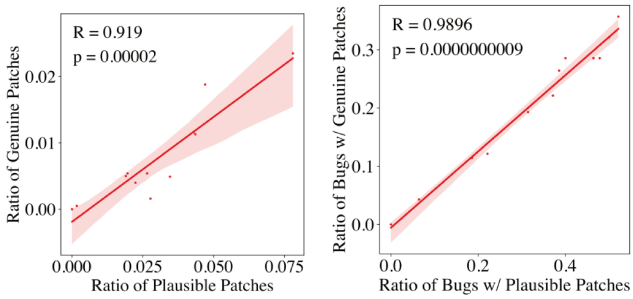

Figure 2: The correlation of plausible and genuine metrics

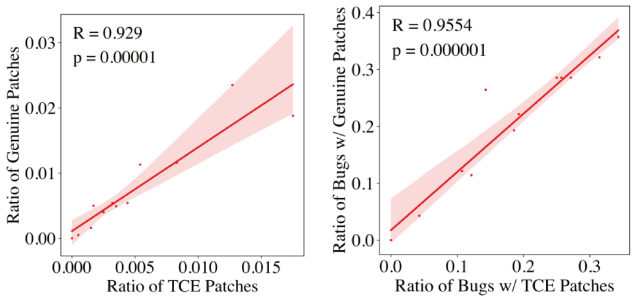

Figure 3: The correlation of TCE and genuine metrics

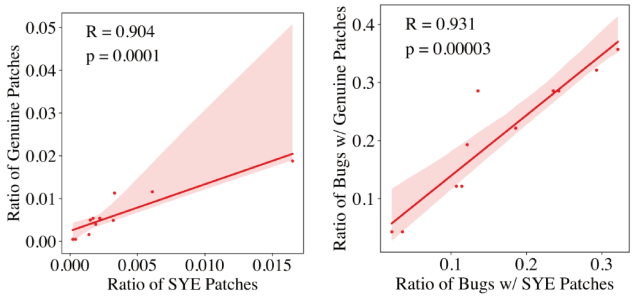

Figure 4: The correlation of SYE and genuine metrics

The overall results are similar to the Defects4J benchmark: TBar has the highest compilability rate, SelfAPR has the most plausible/SYE/TCE patches while AlphaRepair attains the most number of bugs with plausible/SYE/TCE patches. The consistent results across both benchmarks underscore the potential of the mutation-based benchmark as an alternative for large-scale APR evaluations, especially considering the inherent challenges of creating real-world bug benchmarks – the extensive effort required and the limited bug numbers. However, this study concentrates on analyzing APR techniques' performance on simple mutation bugs, deferring the generation of realistic mutation bugs for future research. Additionally, the compilable patch rates on MuBench show a substantial increase for most of the studied APR techniques compared to Defects4J benchmark, indicating that it is easier to generate compilable patches for simpler bugs in the MuBench benchmark.

It is also worth noting that most of the learning-based APR techniques perform better than all the traditional techniques in terms of the number of bugs with TCE patches. Such results show that learning-based APR techniques can handle simple bugs in the MuBench benchmark much better than the traditional ones,

Table 6: The performance of APR techniques on the MuBench benchmark

Metrics	Learning-based Techniques									Traditional Techniques		
	Recoder	SelfAPR	AlphaRepair	RewardRepair	SequenceR	CoCoNut	CURE	Edits	Tufano	Tbar	SimFix	PraPR
# Total Patches	148493	170000	169900	170000	125932	160000	168100	169800	170000	92902	114481	17651
# Compilable Patches	55173 (37.16%)	80334 (47.26%)	72254 (42.53%)	**81055** (47.68%)	40181 (31.91%)	46981 (29.36%)	56950 (33.88%)	3117 (1.84%)	19305 (11.36%)	62472 (**67.25%**)	50546 (44.15%)	N/A
# Plausible Patches	3415 (2.30%)	**15529 (9.13%)**	11305 (6.65%)	8220 (4.84%)	3721 (2.95%)	4136 (2.59%)	4731 (2.81%)	339 (0.20%)	1012 (0.60%)	4088 (4.40%)	785 (0.69%)	1347 (7.63%)
# Bugs w/ Compilable Patches	1654 (97.29%)	1594 (93.76%)	1650 (97.06%)	1678 (98.71%)	1593 (93.71%)	1551 (91.24%)	1518 (89.29%)	698 (41.06%)	878 (51.65%)	**1681 (98.88%)**	1335 (78.53%)	N/A
# Bugs w/ Plausible Patches	803 (47.24%)	1020 (60.00%)	**1125 (66.18%)**	947 (55.71%)	742 (43.65%)	615 (36.18%)	672 (39.53%)	185 (10.88%)	203 (11.94%)	637 (37.47%)	277 (16.29%)	649 (38.18%)
# TCE Patches	855 (0.58%)	**11896 (7.00%)**	2689 (1.58%)	2884 (1.70%)	1017 (0.81%)	1123 (0.70%)	1109 (0.66%)	118 (0.07%)	229 (0.13%)	380 (0.41%)	179 (0.16%)	216 (1.22%)
# SYE Patches	257 (0.17%)	**11273 (6.63%)**	1175 (0.69%)	2171 (1.28%)	342 (0.27%)	777 (0.49%)	643 (0.38%)	100 (0.06%)	107 (0.06%)	341 (0.37%)	161 (0.14%)	N/A
# Bugs w/ TCE Patches	761 (44.76%)	1157 (68.06%)	**1194 (70.24%)**	955 (56.18%)	708 (41.65%)	561 (33.00%)	664 (39.06%)	106 (6.24%)	144 (8.47%)	376 (22.12%)	173 (10.18%)	216 (12.71%)
# Bugs w/ SYE Patches	244 (14.35%)	1128 (66.35%)	**1165 (68.53%)**	915 (53.82%)	342 (20.12%)	542 (31.88%)	643 (37.82%)	100 (5.88%)	107 (6.29%)	341 (20.06%)	161 (9.47%)	N/A

*The percentages in the parentheses denote the ratio of the number of patches to the total number of generated patches for the **# Compilable/Plausible/TCE/SYE Patches** metrics and the ratio of the number of bugs to the total number of bugs for the **# Bugs w/ * Patches** metrics. The largest/highest number/ratio in each row is highlighted in **bold** font.

while such advancement is not significant on the Defects4J benchmark. Another key observation is that, even though the bugs in the MuBench benchmark are relatively simple, the best-performing technique, AlphaRepair, can only produce TCE patches for 70.24% of these bugs, underscoring the potential for enhancement in the current state-of-the-art APR techniques.

> **Finding 4:** Most of the learning-based APR techniques can better handle the simpler bugs in the MuBench benchmark than the traditional ones. However, even the best-performing AlphaRepair only manages to yield TCE patches for 70.24% of the bugs, highlighting the necessity to increase APR techniques' proficiency in fixing simple bugs to make them more applicable in practical scenarios.

4.2.2 Performance in Terms of Mutators. To better understand APR technique performance on different bug types in the MuBench benchmark, we uniformly sample 100 mutants per mutator to collect 700 bugs in total and evaluate the number of bugs with TCE patches for each APR technique. The results are shown in Table 7.

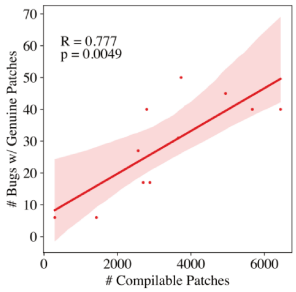

Figure 5: Correlation of compilable and genuine metrics

As shown in Table 7, overall the learning-based APR techniques perform best at fixing LVR bugs, likely due to the simplicity of the LVR mutation, such as substituting numeric values and toggling boolean literals. Meanwhile, the traditional APR techniques perform worse than most of the learning-based techniques on LVR, showing the superiority of learning-based APR techniques in synthesizing literals. On the other hand, the traditional APR techniques outperform most of the learning-based techniques at fixing LOR bugs. A potential reason for this could be that the efficacy of the learning-based techniques on bugs generated by specific mutators is heavily dependent on the recurrence of analogous fix patterns in the training dataset, and LOR appears less frequently in the training data. At the same time, template-based approaches have integrated corresponding templates for such bugs.

Additionally, among the learning-based APR techniques, AlphaRepair, RewardRepair, SequenceR, CURE, Edits and Tufano all have the worst performance for the bugs generated by the mutator SOR. Such results are possibly because the frequency of the fix pattern of operator replacement is relatively low in their training dataset, especially for shift operators. In contrast, SelfAPR is not affected as it is trained on the perturbed correct versions of the buggy programs, which provide more opportunities to see similar fix patterns. Additionally, template-based APR techniques may also overlook corresponding templates for these operators due to their rarity in the data. For example, although TBar has introduced a fix pattern "Mutate Operators", it does not implement the operator mutation for shift and unary operators. As a result, it generates zero TCE patch for ORU and SOR bugs.

We also find that most learning-based and traditional APR techniques (except for SelfAPR and AlphaRepair) struggle to handle the COR mutator. This is because the COR mutator can reduce complex conditional expressions into boolean literals (i.e., `true` and `false`), making it challenging to recover the original conditions. While SelfAPR performs well due to its project-specific training, AlphaRepair performs the best on COR mutants due to the power of large language models in synthesizing boolean expressions.

The results that the traditional and learning-based APR techniques perform better in different bug categories indicate the potential benefits of integrating traditional and learning-based approaches to complement each other's weaknesses. Notably, some existing works [52, 64] have already yielded encouraging outcomes in such a collaborative approach.

> **Finding 5:** Learning-based APR techniques are better at synthesizing literals, while traditional APR techniques are better at replacing rare operators. Such observation suggests the potential advantages of integrating these two types of APR techniques, leveraging their respective strengths to mitigate each other's limitations.

4.3 RQ3: Additional insights for better APR

In this section, we will discuss the additional insights we have gathered in our evaluation, to inspire future APR research.

4.3.1 Patch Compilability.

Correlation between patch compilability and the number of bugs fixed. Intuitively, producing more compilable patches suggests that the APR technique has more valid attempts to generate patches that pass the test suites, leading to more fixed bugs. To validate such an assumption, we conduct a correlation analysis between these two variables. The analysis in Figure 5 reveals a statistically significant correlation ($p < 0.05$) between the number of compilable patches and correctly fixed bugs, validating numerous APR works' efforts to improve patch compilability [18, 61, 62, 66].

Table 7: The performance of APR techniques in terms of different mutators.

Mutator	Learning-based Techniques									Traditional Techniques			#Bugs
	Recoder	SelfAPR	AlphaRepair	RewardRepair	SequenceR	CoCoNut	CURE	Edits	Tufano	Tbar	SimFix	PraPR	
AOR	59	**70**	**70**	59	50	37	37	4	3	21	2	16	100
COR	10	50	**57**	19	8	2	11	0	1	3	4	0	100
LOR	39	55	5	27	16	20	21	1	5	44	15	**76**	100
LVR	74	**84**	74	79	72	62	57	19	17	19	16	12	100
ORU	34	**60**	5	53	15	33	25	1	7	0	2	0	100
ROR	30	68	**74**	46	37	16	37	0	6	38	14	6	100
SOR	39	54	2	9	1	10	8	0	0	0	3	**67**	100

*Each cell represents the number of bugs with TCE patches for each APR technique. The largest number in each row is highlighted in **bold**, and the largest number in each column is underlined.

Table 8: The categories of compilation errors of studied APR techniques on the MuBench benchmark.

Compilation Error	Learning-based Techniques									Traditional Techniques		Total
	Recoder	SelfAPR	AlphaRepair	RewardRepair	Sequencer	CoCoNut	CURE	Edits	Tufano	Tbar	SimFix	
cannot find symbol	11809	27719	63359	33749	22783	26014	16048	31235	13727	8341	48305	303089
... expected	20041	21294	4695	11272	13192	11038	19183	34533	58534	134	3552	197468
illegal start of expression	6668	12136	4181	9998	7945	6963	19737	77050	28968	32	3904	177582
unclosed character literal	0	903	381	331	263	42373	16018	235	0	0	592	61096
incompatible types	9391	2157	2102	3027	7144	1552	3587	2403	6531	904	679	39477
bad operand ... for (binary|unary) operator	10587	2858	1682	2714	7376	2096	3958	305	2529	3202	255	37562
not a statement	1422	3285	1598	7234	5417	639	2683	3735	4373	28	179	30593
incomparable types	6677	1610	1969	2115	5373	1818	3490	0	2858	1888	258	28056
(method|constructor) ... cannot be applied to given types	4587	1707	2065	1860	3879	2076	3726	261	4703	1903	402	27169
illegal start of type	2613	2320	786	2485	301	58	5318	5888	5779	1	5	25554
Total (Semantic Error)	43051 (58.34%)	36051 (47.44%)	71177 (85.94%)	43465 (58.12%)	46555 (63.19%)	33556 (35.46%)	30809 (32.86%)	34204 (21.98%)	30348 (23.71%)	16238 (98.81%)	49899 (85.84%)	435353
Total (Syntactic Error)	30744 (41.66%)	39938 (52.56%)	11641 (14.06%)	31320 (41.88%)	27118 (36.81%)	61071 (64.54%)	62939 (67.14%)	121441 (78.02%)	97654 (76.29%)	195 (1.19%)	8232 (14.16%)	492293
Total	73795	75989	82818	74785	73673	94627	93748	155645	128002	16433	58131	927646

Finding 6: The patch compilability rate is highly correlated to the number of bugs being correctly fixed, suggesting that enhancing the patch compilability may lead to more valid fix attempts and potentially more fixed bugs.

Strategies used to improve compilability. Among NMT-based APR techniques, Recoder, SelfAPR, RewardRepair, and CURE have the highest compilability rates on both benchmarks, each employing a specific strategy to improve compilability. Specifically, RewardRepair uses a compilability discriminator, SelfAPR includes uncompilable perturbed programs in its training dataset to avoid generating uncompilable patches, Recoder selects compatible identifiers, and CURE utilizes pre-trained language models with a code-aware beam-search strategy. In contrast, TBar maintains high compilability by employing template designs likely to preserve syntactic validity. Despite using varied strategies, learning-based techniques lag behind TBar's compilability rate in all benchmarks, suggesting a need for future research to boost patch compilability in learning-based APR techniques.

Finding 7: The various compilability improvement strategies adopted by studied learning-based APR techniques cannot surpass the compilability achieved by straightforward template-based mutation. This suggests improving the patch compilability of learning-based APR techniques through more effective syntactic-validity-preserving strategies.

Compilation error categorization. To gain insights into the factors limiting the studied techniques in generating compilable patches, we analyze the compilation errors of uncompilable patches generated on the large-scale MuBench benchmark. Table 8 lists the top 10 errors with a heatmap indicating their frequency. Note that the data for PraPR is not included as PraPR generates patches at bytecode-level, eliminating the need for compilation. The Java compiler (javac) can pinpoint both syntactic and semantic errors, with the latter marked in green background color in the first column of the table. Note that syntactical validity is a basic requirement for compilability, and comparing the proportion of the syntactically valid patches to the semantically valid ones can help gauge the progress of an APR method in improving patch compilability.

The common error across all techniques is the *cannot find symbol* error, indicating a lack of context understanding for accessible identifiers. AlphaRepair and TBar show the lowest syntactic error rates (14.06% and 1.19%) among learning-based and traditional techniques, thanks to large language model basis and AST-based mutation operations. Notably, with a syntactical error rate of 14.16%, SimFix ranks third, closely trailing AlphaRepair. Recoder and RewardRepair both have more than 40% syntactical errors. Despite Recoder's effort to pick type-compatible identifiers, it still faces 9391 (12.73%) *incompatible types* issues. On the other hand, CoCoNut and CURE generate a large number of uncompilable patches due to *unclosed character literal* errors, reflecting wasted patch exploration, even though CURE employs a pre-trained language model. Moreover, Edits and Tufano have major syntactic issues like missing symbols and incorrect Java syntax usage, possibly due to their smaller training datasets.

Finding 8: Examining compilation errors of generated patches can reveal bottlenecks in generating compilable patches. Traditional APR techniques are generally more effective at avoiding syntactic errors in patches than learning-based APR techniques. The most frequent compilation error *cannot find symbol* indicates that many state-of-the-art APR techniques struggle to generate valid identifiers.

4.3.2 Duplicate and No-op Patches. Besides patch compilability, we examine duplicate and no-op patches generated by the studied APR techniques. Duplicate patches are syntactically equivalent to

Table 9: The statistic of duplicate/no-op patches on the MuBench benchmark.

Metrics	Learning-based Techniques									Traditional Techniques	
	Recoder	SelfAPR	AlphaRepair	RewardRepair	Sequencer	CoCoNut	CURE	Edits	Tufano	Tbar	SimFix
# Duplicate patches	2902 (1.95%)	**67453 (39.68%)**	443 (0.26%)	37284 (21.93%)	5 (0.00%)	4654 (2.91%)	57 (0.03%)	136 (0.08%)	25836 (15.20%)	21 (0.02%)	4830 (4.22%)
# No-op patches	996 (0.67%)	**5119 (3.01%)**	877 (0.52%)	4020 (2.36%)	750 (0.60%)	1746 (1.09%)	1162 (0.69%)	402 (0.24%)	345 (0.20%)	94 (0.10%)	4 (0.00%)
# Bugs with duplicate patches	887 (52.18%)	**1700 (100.00%)**	47 (2.76%)	1685 (99.12%)	3 (0.18%)	741 (43.59%)	13 (0.76%)	31 (1.82%)	1169 (68.76%)	5 (0.29%)	215 (12.65%)
# Bugs with no-op patches	817 (48.06%)	928 (54.59%)	839 (49.35%)	**1479 (87.00%)**	750 (44.12%)	1277 (75.12%)	1162 (68.35%)	402 (23.65%)	345 (20.29%)	94 (5.53%)	4 (0.24%)

previously generated patches, while no-op patches are syntactically equivalent to the original buggy program, wasting computational resources without improving patch diversity. We employ the SYE metrics, i.e., comparing tokenization results to determine syntactical equivalence, to identify such patches generated on the MuBench benchmark. The analysis results are shown in Table 9.

The results show that SelfAPR has the most significant patch duplication issue with about 40% duplicate patches, and it generates at least one duplicate patch for each bug. RewardRepair and Tufano also have high duplication rates of 21.93% and 15.20% respectively, with 99.12% and 68.71% bugs associated with duplicate patches. SelfAPR leads with a 3.01% rate of no-op patches, while RewardRepair has the highest ratio of bugs with no-op patches at 87.00%. Although duplicate and no-op patches may not directly hinder the effectiveness of APR techniques, they can consume computational resources and limit the exploration of diverse patches. Identifying such patches beforehand can save costs on unnecessary patch validation and manual correctness checking.

> **Finding 9:** Four out of nine learning-based APR techniques incur more than 40% of bugs having duplicate/no-op patches, highlighting the opportunity for corresponding optimization techniques to enhance the efficiency and reliability of learning-based APR techniques.

4.3.3 Implementation Issues. Our extensive evaluation aids in identifying implementation issues in the studied APR techniques, especially on the large-scale MuBench benchmark. There are mainly two types of issues: 1) some APR techniques underperform, e.g., failing to fix certain bugs that should have been fixed, and 2) certain techniques crash when trying to repair certain bugs, exposing potential implementation issues.

Inferior performance. For instance, Recoder aims to use only feasible identifiers—identifiers accessible in the local context and meeting type constraints—to substitute original identifiers. Yet, as Section 4.3.1 notes, it leads to 11809 and 9391 compilation errors from *cannot find symbol* and *incompatible types*, accounting for 28.73 % of all its compilation errors. To probe the cause, we examine the error-inducing patches generated by Recoder. Listing 1 shows a patch by Recoder, replacing literals '`0`' and '`9`' with "`null`", and `str.charAt(i)` with `str.getDurationMillis(i)`, leading to compilation errors since > cannot be applied to a String, and `getDurationMills` is not a String method. Upon inspecting Recoder's source code, we find that it does not always fully guarantee the feasibility of identifiers, i.e., well-typedness and accessibility.

In another case, TBar, designed to include the *Mutate Literal Expression* mutator (altering literals to other correspondingly-typed literals or expressions), merely fixes 19 out of 100 LVR bugs. For instance, it fails to fix the simple bug illustrated in Listing 2. Upon reviewing its generated patches, we find that instead of substituting with other integer values, it solely substitutes the integer literals with double/float literals of the same value. Additionally, we found that TBar does not implement the functionality to replace String/Character literals, although it is expected to do so. Similarly, PraPR only fixes 12 LVR bugs, attributed to its restrictive literal replacement pattern, such as only mutating int literal `i` to either `0` or `i+1` for numeric literals mutation.

Listing 1 Example patch of Recoder

```
...
- if(str.charAt(i) > '0' && str.charAt(i) <= '9'){
+ if(((str.charAt(i) > "null") && (str.getDurationMillis(i) <=
↪  "null"))){
...
```

Listing 2 Example LVR bug that TBar failed to fix

```
...
- if (contains(value, index + 2, 1, "I", "E", "H") &&
+ if (contains(value, index + 2, -1, "I", "E", "H") &&
...
```

Listing 3 Buggy pre-processing code snippet of Recoder

```
# Recoder testone.py
if mode == 1:
    # aftercode represents the subsequent context
    aftercode = oldcode + aftercode
lines = aftercode.splitlines()
if 'throw' in lines[0] and mode == 1: # IndexError
    for s, l in enumerate(lines):
...
```

Crashing in repairing. Some techniques crash while attempting to fix certain bugs. For example, Listing 3 shows a faulty code snippet from the pre-processing script of Recoder, aiming to extract the context following the buggy line within the same method. However, if the buggy line encompasses the entire method, such as `int sum(int a, int b){return a+b;}`, the script will crash as `aftercode` and `lines` turn empty, causing `lines[0]` to trigger `IndexError`. This bug was confirmed and fixed by its authors. In total, we have identified seven issues in AlphaRepair, SequenceR, SimFix, TBar, Recoder, and CoCoNut, with five confirmed and fixed by their authors. The list of issues can be found in our artifact [3].

> **Finding 10:** Our multi-dimensional evaluation and analyses have revealed seven implementation issues (with five confirmed and fixed by corresponding authors) in six studied APR techniques.

5 THREATS TO VALIDITY

The threats to external validity mainly lie in the limited number of benchmarks used for the evaluation and the generalizability of the evaluation results. Therefore, besides the Defects4J benchmark, we also build the mutation-based MuBench benchmark that contains 1,700 bugs generated by various mutators, larger than prior individual benchmarks for APR evaluation. Another concern is that

the MuBench benchmark constructed with 7 mutators might not accurately represent real-world bugs, thus results from it may not genuinely reflect the capabilities of the assessed APR techniques. To address this concern, in this paper, we utilize MuBench benchmark results solely to analyze the characteristics of the techniques evaluated. Another threat lies in the potential data leakage issue in learning-based APR techniques. To mitigate this threat, we carefully review the studied techniques to ensure they handled the data leakage issue properly. Moreover, the MuBench benchmark we create further alleviates such an issue by introducing new bugs through program mutation.

Threats to internal validity mainly lie in the usage of APR techniques and the manual patch inspection process. Thus, we meticulously adhere to the instructions provided in each APR technique's `README` file and proactively communicate with the authors if any procedure is unclear. Additionally, we ensure the accuracy of manual patch inspection by engaging two authors experienced in Java development to independently verify the correctness of the patches.

The threats to the construct validity mainly lie in the metrics used. Thus, we include the metrics that have been used in previous works such as compilability, plausibility, and genuineness metrics. For newly introduced metrics , i.e., SYE and TCE metrics, we perform correlation analyses to show that they are highly correlated to the genuineness metrics.

6 CONCLUSION

In this study, we comprehensively evaluate nine learning-based and three traditional APR techniques employing the Defects4J benchmark, alongside the substantial mutation-based MuBench benchmark consisting of 1,700 artificial bugs. Our analyses of the 1,814,652 generated patches utilize multi-dimensional metrics including compilability, plausibility, SYE, TCE, and genuineness metrics, which allow for an in-depth understanding of the capabilities and areas of improvement for the studied APR techniques. Our comprehensive evaluation leads to multiple findings. For instance, LLM-based APR is generally less prone to overfitting compared to NMT-based and traditional techniques. TCE metrics could be used as cost-efficient alternatives for genuineness metrics for large-scale APR evaluations. All studied learning-based techniques lag behind template-based techniques in producing compilable patches. Additionally, many studied learning-based techniques suffer from the issue of generating duplicate/no-op patches which could burden their repair effectiveness. We also provide valuable insights for future research, including: the need for equitable evaluation settings (e.g., uniform candidate patch numbers and patch generation mechanisms); the potential benefits of integrating traditional and learning-based techniques to capitalize on their respective strengths in fixing different types of bugs; and the importance of improving patch compilability to yield more valid fix attempts and correct bug fixes. Moreover, our study reveals seven implementation issues within the studied APR techniques, and five of them were confirmed and subsequently fixed by the respective authors.

DATA AVAILABILITY

Our replication package is available at https://github.com/mutrepair/BenchmarkingAPR and [3].

ACKNOWLEDGMENTS

This work was partially supported by NSF grant CCF-2131943, CCF-1763788 and CCF-1956374, as well as by Kwai Inc. We thank all the reviewers for their insightful comments. We also thank Chunqiu Steven Xia and Yifeng Ding from UIUC, He Ye and Zimin Chen from KTH, Wenkang Zhong from Nanjing University, and Qihao Zhu from Peking University for their help.

REFERENCES

[1] 2023. ANTLR. https://www.antlr.org/.
[2] 2023. ASM. https://asm.ow2.io/.
[3] Yicheng Ouyang, Jun Yang, and Lingming Zhang. 2024. ISSTA'24 Artifact for "Benchmarking Automated Program Repair: An Extensive Study on Both Real-World and Artificial Bugs". https://doi.org/10.5072/zenodo.45711
[4] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. 2017. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 511–522. https://doi.org/10.1145/3106237.3106280
[5] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2022. CODIT: Code Editing With Tree-Based Neural Models. *IEEE Transactions on Software Engineering* 48, 4 (2022), 1385–1399. https://doi.org/10.1109/TSE.2020.3020502
[6] Lingchao Chen, Yicheng Ouyang, and Lingming Zhang. 2021. Fast and Precise On-the-Fly Patch Validation for All. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1123–1134. https://doi.org/10.1109/ICSE43902.2021.00104
[7] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based program repair without the contracts. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 637–647. https://doi.org/10.1109/ASE.2017.8115674
[8] 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. *IEEE Transactions on Software Engineering* 47, 09 (sep 2021), 1943–1959. https://doi.org/10.1109/TSE.2019.2940179
[9] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 449–452. https://doi.org/10.1145/2931037.2948707
[10] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. 2009. Generating Fixes from Object Behavior Anomalies. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, USA, 550–554. https://doi.org/10.1109/ASE.2009.15
[11] Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J. Hellendoorn. 2021. Patching as translation: the data and the metaphor. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/3324884.3416587
[12] 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 *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 302–313. https://doi.org/10.1145/3338906.3338911
[13] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2018. Automatic software repair: a survey. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1219. https://doi.org/10.1145/3180155.3182526
[14] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 19–30. https://doi.org/10.1145/3293882.3330559
[15] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-based program repair using SAT. In *Tools and Algorithms for the Construction and Analysis of Systems: 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 17*. Springer, 173–188.
[16] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2020. Inferring program transformations from singular examples via big code. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 255–266. https://doi.org/10.1109/ASE.2019.00033

[17] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1430–1442. https://doi.org/10.1109/ICSE48619.2023.00125

[18] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[19] René Just. 2014. The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 433–436. https://doi.org/10.1145/2610384.2628053

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

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

[22] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Vancouver, BC, Canada) *(SPLASH Companion 2017)*. Association for Computing Machinery, New York, NY, USA, 55–56. https://doi.org/10.1145/3135932.3135941

[23] 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 *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 102–113. https://doi.org/10.1109/ICST.2019.00020

[24] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/3293882.3330577

[25] 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 *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 615–627. https://doi.org/10.1145/3377811.3380338

[26] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 166–178. https://doi.org/10.1145/2786805.2786811

[27] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[28] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. 2014. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 227–238. https://doi.org/10.1145/2594291.2594337

[29] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/3395363.3397369

[30] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: an automated class mutation system: Research Articles. *Softw. Test. Verif. Reliab.* 15, 2 (jun 2005), 97–133.

[31] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 468–478. https://doi.org/10.1109/SANER.2019.8667991

[32] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. SapFix: Automated End-to-End Repair at Scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 269–278. https://doi.org/10.1109/ICSE-SEIP.2019.00039

[33] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Softw. Engg.* 22, 4 (aug 2017), 1936–1964. https://doi.org/10.1007/s10664-016-9470-4

[34] Matias Martinez and Martin Monperrus. 2016. ASTOR: a program repair library for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 441–444. https://doi.org/10.1145/2931037.2948705

[35] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[36] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781. https://doi.org/10.1109/ICSE.2013.6606623

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

[38] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 936–946. https://doi.org/10.1109/ICSE.2015.103

[39] 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

[40] Jibesh Patra and Michael Pradel. 2021. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 906–918. https://doi.org/10.1145/3468264.3468623

[41] Karl Pearson. 1895. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London* 58, 347-352 (1895), 240–242. https://doi.org/10.1098/rspl.1895.0041

[42] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. 2009. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 87–102. https://doi.org/10.1145/1629575.1629585

[43] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/2568225.2568254

[44] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. 2018. Bugs.jar: a large-scale, diverse dataset of real-world Java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 10–13. https://doi.org/10.1145/3196398.3196473

[45] David Schuler and Andreas Zeller. 2009. Javalanche: efficient mutation testing for Java *(ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 297–298. https://doi.org/10.1145/1595696.1595750

[46] Ridwan Shariffdeen, Martin Mirchev, Yannic Noller, and Abhik Roychoudhury. 2023. Cerberus: A Program Repair Framework. In *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 73–77. https://doi.org/10.1109/ICSE-Companion58688.2023.00028

[47] Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. 2023. Learning to Construct Better Mutation Faults. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 64, 13 pages. https://doi.org/10.1145/3551349.3556949

[48] Michele Tufano, Jason Kimko, Shiya Wang, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, and Denys Poshyvanyk. 2020. DeepMutation: a neural mutation tool. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 29–32. https://doi.org/10.1145/3377812.3382146

[49] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An Empirical Investigation into Learning

Bug-Fixing Patches in the Wild via Neural Machine Translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 832–837. https://doi.org/10.1145/3238147.3240732

[50] Vasudev Vikram, Isabella Laybourn, Ao Li, Nicole Nair, Kelton OBrien, Rafaello Sanna, and Rohan Padhye. 2023. Guiding Greybox Fuzzing with Mutation Testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 929–941. https://doi.org/10.1145/3597926.3598107

[51] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Trento, Italy) *(ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/1831708.1831716

[52] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 172–184. https://doi.org/10.1145/3611643.3616271

[53] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233

[54] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[55] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 959–971. https://doi.org/10.1145/3540250.3549101

[56] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *ArXiv* abs/2304.00385 (2023). https://api.semanticscholar.org/CorpusID:257913714

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

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

[59] Jun Yang, Yuehan Wang, Yiling Lou, Ming Wen, and Lingming Zhang. 2023. A Large-Scale Empirical Review of Patch Correctness Checking Approaches. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1203–1215. https://doi.org/10.1145/3611643.3616331

[60] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the QuixBugs benchmark. *Journal of Systems and Software* 171 (2021), 110825. https://doi.org/10.1016/j.jss.2020.110825

[61] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2023. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 92, 13 pages. https://doi.org/10.1145/3551349.3556926

[62] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1506–1518. https://doi.org/10.1145/3510003.3510222

[63] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. 2016. Predictive mutation testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 342–353. https://doi.org/10.1145/2931037.2931038

[64] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. Gamma: Revisiting Template-Based Automated Program Repair Via Mask Prediction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 535–547. https://doi.org/10.1109/ASE56229.2023.00063

[65] Wenkang Zhong, Hongliang Ge, Hongfei Ai, Chuanyi Li, Kui Liu, Jidong Ge, and Bin Luo. 2023. StandUp4NPR: Standardizing SetUp for Empirically Comparing Neural Program Repair Systems. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 97, 13 pages. https://doi.org/10.1145/3551349.3556943

[66] 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 *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 341–353. https://doi.org/10.1145/3468264.3468544

Received 16-DEC-2023; accepted 2024-03-02