The Future Can't Help Fix The Past: Assessing Program Repair In The Wild

Vinay Kabadi[1], Dezhen Kong[2], Siyu Xie[3], Lingfeng Bao[2*],
Gede Artha Azriadi Prana[4], Tien-Duy B. Le[4], Xuan-Bach D. Le[1], David Lo[4]

[1]School of Computing and Information Systems, The University of Melbourne, Australia
[2]School of Computer and Computing Science, Zhejiang University, China
[3]College of Computer Science and Technology, Zhejiang University, China
[4]School of Information Systems, Singapore Management University, Singapore
vkabadi@student.unimelb.edu.au, timkong@zju.edu.cn, 3140102422@zju.edu.cn, lingfengbao@zju.edu.cn,
arthaprana.2016@phdis.smu.edu.sg, btdle.2012@phdis.smu.edu.sg, bach.le@unimelb.edu.au, davidlo@smu.edu.sg

***Abstract*—Automated program repair (APR) has been gaining ground with substantial effort devoted to the area, opening up many challenges and opportunities. One such challenge is that the state-of-the-art repair techniques often resort to incomplete specifications, e.g., test cases that witness buggy behavior, to generate repairs. In practice, bug-exposing test cases are often available when: (1) developers, at the same time of (or after) submitting bug fixes, create the tests to assure the correctness of the fixes, or (2) regression errors occur. The former case – a scenario commonly used for creating popular bug datasets – however, may not be suitable to assess how APR performs in the wild. Since developers already know where and how to fix the bugs, tests created in this case may encapsulate knowledge gained *only after* bugs are fixed. Thus, more effort is needed to create datasets for more realistically evaluating APR.**

We address this challenge by creating a dataset focusing on bugs identified via continuous integration (CI) failures – a special case of regression errors – wherein bugs happen when the program after being changed is re-executed on the existing test suite. We argue that CI failures, wherein bug-exposing tests are created before bug fixes and thus assume no prior knowledge of developers on the bugs to be involved, are more realistic for evaluating APR. Toward this end, we curated 102 CI failures from 40 popular real-world software on GitHub. We demonstrate various features and the usefulness of the dataset via an evaluation of five well-known APR techniques, namely GenProg, Kali, Cardumen, RsRepair and Arja. We subsequently discuss several findings and implications for future APR studies. Overall, experiment results show that our dataset is complementary to existing datasets such as Defect4J in realistic evaluations of APR.

[1] ***Index Terms*—Program Repair, Benchmark**

I. Introduction

Bug fixing is notoriously difficult and costly in terms of time and effort spent by developers [6], [55]. This is largely due to the fact that this process still rests entirely on human to manually repair bugs in practice. Thus, automated program repair (APR) techniques that can help efficiently and effectively automate bug fixing would be of tremendous value. The once futuristic idea of APR has been brought closer to reality by substantial recent research effort devoted to the area [7], [16], [19], [21], [22], [28], [34], [36], [40], [41], [62], [63]. Two main families of APR include heuristics- vs semantics-based approaches, each of which generates and traverses the search space for repairs in a different way.

Recent pragmatic advancements in APR have opened up many challenges and opportunities. One such challenge is that most state-of-the-art APR approaches hinge on the use of incomplete specifications, e.g., test cases with at least one of which is failing and thus exposing buggy behavior, to generate repairs [16], [18], [26], [28], [34], [40], [41], [62]. Generally, APR techniques first dynamically analyse the program under repair using tests to localize potentially buggy elements. They then generate and traverse a huge search space for repairs, partially inferred through the performed dynamic analysis. The effectiveness of APR thus largely depends on the information obtained from the dynamic analysis, which in turn relies on the existence and various properties of fault-revealing test cases.

In practice, fault-revealing test cases are often obtained when: (1) developers, at the same time of (or after) submitting bug fixes, produce the tests for assuring the correctness of the fixes, or (2) regression errors occur via continuous integration (CI) testing, wherein existing tests, that previously pass, now fail. We refer the test cases obtained from the former as *future* test cases and from the latter as *existing* test cases. Existing datasets commonly used for validating APR techniques such as Defects4J [14], Bugs.jar [51], and Bears [37], are integrated with *future* test cases. We argue that using future test cases may not be able to accurately estimate how well APR would work in practice. That is, since developers already perceived where and how the bugs can be fixed, tests created in this case may be too specific to the bugs and unduly provide implicit help to enhance APR. Such help is unavailable in practice when APR needs to fix bugs *before* developers fix them and create suitable test cases to ensure the absence of the bugs. The latter case (regression bugs) on the other hand, assumes no prior knowledge of developers on the bugs since fault-revealing tests are created before the bugs are fixed, and thus is better able to estimate how APR would perform in practice. Despite their differences, curating datasets in either case is a

[1]* Corresponding authors

2576-3148/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSME58846.2023.00017

TABLE I
FUTURE TEST CASES IN EXISTING BENCHMARK DATASETS

Benchmark	Total Bugs	Bugs with future test cases
Defects4J	395	381
Bugs.jar	1130	1064
Bears	251	232

non-trivial task, which could easily take up months or even years of strenuous engineering effort [14], [29], [57].

A recent study [31] as shown in Table I demonstrates that more than 92% of the bugs in popular data sets used to evaluate APR are integrated with *future* test cases, i.e., bug-witnessing test cases that are created after the bug is reported. Hence, APR evaluations on these data sets may not fully reflect how APR works in the wild. APR assessment should explicitly differentiate experiments that are valid *in-the-lab* from those that would approximate *in-the-wild* performance. Liu et al. [31] shows that 381 out of 395 Defects4J bugs are patched and validated with future test cases. When such test cases are dropped from the test suites, various APR tools could only fix fewer than 6 bugs with correct patches. Hence, we have built a dataset that confirms the bias shown to exist in existing datasets. Our dataset is complementary to existing datasets in that it does not involve *future test cases* for fixed bugs – wherein no prior knowledge on how the bug is fixed is involved in constructing fault-revealing tests. This helps to clearly judge which techniques are effective for what classes of bugs.

Recently, Bugswarm [56] has been proposed to provide a dataset of bugs in the wild that do not involve future test cases. It is a continuously growing set of failing and passing versions of real-world, open-source systems, in Java and Python [56]. Although, it has a large set of bugs, only a part of them can be used for APR purposes. The dataset contains bugs such as code bugs, error in tests cases, build issues, etc. but current APR techniques mainly target behavioral bugs that present in the program source code. A study by Durieux and Abreu shows limitations in the benchmark, e.g., only 50 Java bugs and 62 python bugs out of 3091 bugs are suitable to evaluate automatic fault localization or program repair techniques [11].

We argue that evaluating APR on less realistic benchmarks or smaller dataset will not only result in less reliable outcomes, but also can risk setting back the advancement of the whole field. To address this challenge, we propose to create a new dataset specifically designed for evaluating APR on larger dataset of Java regression bugs. In particular, our dataset includes 102 java regression bugs from 40 popular real-world large software written in Java programming language on GitHub. Our dataset has 29 projects in common with Bugswarm but all the bugs are different. The dataset has only regression bugs, featuring no future test cases hence contains no prior knowledge on the bugs. It includes various bugs from a wide range of systems, which are well-categorized into six defect categories. We focus on Java since there is an increasing number of APR tools proposed recently that target Java [19], [26], [39], [61]–[63].

TABLE II
DATASET COMPARISON BETWEEN DEFECTS4J AND OUR DATASET

Defects4J - 356 Bugs		**Our dataset - 102 bugs**	
Bugs%	**Lines Modified**	**Bugs%**	**Lines Modified**
62	0 - 5	91	0 - 5
33	6 - 20	9	6 - 20
5	20 - 50	5	20 - 50
0	50+	2	50+

We demonstrate the features and usefulness of our dataset via a comparative study with the Defects4J benchmark [2], which has been extensively referred in the space of APR. We find that Defects4J is a suitable basis for our study for two reasons. First, Defects4J contains real bugs from large real-world Java programs, and has recently become a popular benchmark for APR evaluations [24], [61]–[63]. Second, we want to investigate the differences between a dataset possessing the envisioned features (i.e., our dataset) versus another that does not possess the features (e.g., Defects4J), and their possible impacts on the effectiveness of APR techniques. Toward that end, we seek to answer the following research questions:

- **RQ1** *How different are failure-exposing test cases in our dataset than those included in Defects4J?* To answer this research question, we compare some statistics that characterize failure-exposing test cases in our dataset and Defects4J, and highlight noticeable differences.
- **RQ2** *Are there differences in automated program repair effectiveness evaluated using our dataset as compared to Defects4J?* To answer this research question, we run APR solutions on all bugs in our dataset and compare their effectiveness with the ones that are reported in the literature.

In RQ1, we compute a number of statistics to assess characteristics of failure-exposing test cases in our dataset and Defects4J [14]. Our dataset has more than *six times* more failure-exposing test cases than those of Defects4J, and failure-exposing test cases in our dataset are more than *three times* less localizable, i.e., less chance to localize the root causes of bugs, and *close to 70%* less similar, e.g., less chance to provide fixing ingredients, to bug fix patches than those of Defects4J. Table II shows the statistics on the number of lines of code changes of bug fixes in Defects4J and our dataset. From Table II, we can see that the majority of bug fixes in both our dataset and Defects4J have fewer than five lines of code changes.

For RQ2, we run GenProg, Kali, Cardumen implemented in Astor [39], Arja [65] and RsRepair [48] on our dataset. We choose these tools because the same tools were also applied on Defects4J in a recent prior study [8], [38], [65] and we wish to compare the performance of APR tools against our dataset. We did not use semantics-based APR tools in our study such as SemFix [42], DirectFix [40], Angelix [41], and S3 [26]. These tools use symbolic analysis which often requires manually writing models for system calls whose source codes are not available. We tried to configure ACS [62] on our dataset, but

[2] https://github.com/rjust/defects4j/releases/tag/v1.0.0

failed to successfully do so because ACS hard-coded their configurations for only Defects4J.

The results of RQ2 suggest that correct patch generation rate on our benchmark is *eight times* lower as compared to results reported in [8], [38].This low result is despite the fact that our dataset has fewer lines of code modified per bug compared to the Defects4J as shown in Table II. The mean of Defects4J and our dataset are 6.43 and 5.05 lines modified per bug respectively.

In summary, the contributions of this work are as follow:

1) We provide a dataset of 102 real CI failures from 40 large real-world Java programs. Our dataset displays several features that are specifically suitable for more realistic APR evaluation. We save the community months of engineering efforts by making our dataset publicly available via GitHub.[3]
2) We demonstrate various features and the usefulness of our proposed dataset by means of an evaluation of five well-known repair techniques namely GenProg, Kali, Cardumen, Arja and RsRepair. Experiment results suggest that there is a lot of room for improvement for future APR techniques. Our dataset is one of the first steps to open up such opportunities.

The remaining of the paper is organized as follow. Section II presents background on program repair and examples that motivate the construction of our dataset. Section III explains the methodology to construct our dataset, followed by Section IV that describes several statistics of the constructed dataset. Section V presents evaluation results for our research questions, followed by discussions in Section VI. Section VII presents related work. Section VIII concludes the paper.

II. Background

In this section, we explain some popular APR techniques and empirical studies. We then briefly describe motivating examples for our newly proposed dataset.

A. Automated Program Repair

a) Program repair.: Repair techniques can generally be divided into two families: search-based vs semantics-based, classified by the ways they generate and traverse the search space for repairs. Search-based approaches generate a large number of repair candidates and employ search or other heuristics to identify correct repairs among them. Semantics-based approaches extract semantics constraints from test suites, and synthesize repairs that satisfy the extracted constraints. GenProg [28] is an early search-based APR tool that uses genetic programming to search for a repair that causes a program to pass all provided tests cases among a possibly huge pool of repair candidates. GenProg targets generic bugs by using general mutation operators to generate repair candidates such as statement deletion, replacement, and append. In a similar vein, Kali [50] attempts to generate *test-suite-adequate repair*[4] by only deleting statements. It has been shown that Kali, despite being simple, generates as many correct patches as prior repair systems such as GenProg [50]. Recently, Arja [65] a new GP based repair approach for automated repair of Java programs presents a novel lower-granularity patch representation that properly decouples the search subspaces of likely-buggy locations, operation types and potential fix ingredients, enabling GP to explore the search space more effectively. RsRepair [48], can automatically generate patches for faulty programs by using purely random search algorithm and also comes with an adapted test case prioritization technique to speed up the patch validation process.

b) Studies in APR: Qi et al. [50] manually analyze the correctness of bug fixes generated by GenProg and its variants using ManyBugs benchmark [29]. They manually write additional test cases to augment existing test suites, and show that the majority of generated patches is incorrect since the patches do not pass the additional test cases. A later study by Smith et al. [52] refer this problem as *overfitting*, and confirm that many of patches generated by search-based repair techniques such as GenProg and the likes actually pass all tests used for repair but do not generalize to other independent test suites. In a similar vein, Le et al. [25] show that semantics-based repair techniques are also no exception to the overfitting issue. Particularly, they show that a large fraction (up to 90%) of patches generated by semantics-based APR are overfitting by using the IntroClass [29] and Codeflaws [53] datasets, each of which contains small programs and independent test suites for automatic patch correctness assessment. Martinez et al. [38] empirically study several APR techniques (including their own technique namely Nopol [63]) in both semantics- and search-based families on Defects4J dataset [14]. The correctness of machine-generated patches were manually assessed by the authors of the paper. Yi et al. [64] study several APR techniques on introductory programming assignments. These studies, despite being conducted rigorously, used datasets that either do not contain bugs from real-world large programs (e.g., IntroClass and Codeflaws), or fully satisfy the desirable features that we discussed in Section 1 (e.g., Defects4J and ManyBugs).

B. Motivating Examples

To motivate our proposed dataset, we start by showing an example of a fault-revealing (i.e., failing) test case created at the same time as or after a bug is fixed (i.e., *future test case*). We then compare it with another fault-revealing test case created before a bug is even identified (i.e., *existing test case*). We highlight some noteworthy differences of the two cases.

Future test case – A bug fix in Apache Commons Math[5] and its fault-revealing test case, which are included as part of the Defects4J dataset, are shown in Figures 1 and 2, respectively. The bug fix and its failing test are submitted *at the same time* by developers. We can note that the failing test is very specific

[3] https://github.com/CI-Bugs/Repo1

[4] A test-suite-adequate repair is a bug fix that makes all test cases pass. Such repairs are not guaranteed to be correct though as a test suite often does not cover all possible scenarios.

[5] http://commons.apache.org/proper/commons-math/

to the buggy method, that is the `linearCombination` method in class `MathArrays` – see line 4 in Figure 2, and line 2 in Figure 1. Aside this method, the failing test includes invocation to no other methods in Apache Commons Math. Additionally, the fault-revealing test shares noticeable commonalities with the fix (i.e., "a[0] * b[0]").

```
1  // In class MathArrays
2  public static double linearCombination(double[] a, double[]
       b) throws ... {
3      ...
4  + if (len == 1) {
5  +   // Revert to scalar multiplication
6  +   return a[0] * b[0];
7  + }
8    ...
9  }
```

Fig. 1. A bug fix created by developers for bug MATH-1005 in Apache Commons Math, which corresponds to Math3 in Defects4J dataset. The failing test that exposes this bug is depicted in Figure 2.

```
1  public void testArray() {
2      final double [] a = { 1.23456789 } ;
3      final double [] b = { 98765432.1 } ;
4      Assert.assertEquals(a[0] * b[0], MathArrays.
           linearCombination(a, b), 0d) ;
5  }
```

Fig. 2. A failing test of bug Math3 in Defects4J dataset

Existing Test case – We now show an example of a bug fix for a continuous integration (CI) failure appearing for HikariCP[6] and its fault-revealing test case, which are included in our dataset. The bug fix and its failing test case are shown in Figures 3 and 4, respectively. We can note that the test case does not directly invoke the buggy method (i.e., `closeOpenStatements`) but is invoked by a sub procedure internally. The test case invokes many other non-buggy methods in HikariCP. Additionally, the fix and the fault-revealing test case do not share any noticable commonalities.

```
// In class ConnectionProxy
private final boolean closeOpenStatements() {
  final int size = openStatements.size();
- if (size <= 0) {
+ if (size > 0) {
     boolean success = true;
     for (int i = 0; i < size; i++) {
     ...
}
```

Fig. 3. A bug fix for a HikariCP's bug identified via a continuous integration failure

```
public void testAutoStatementClose() throws SQLException {
  Connection connection = ds.getConnection();
  Assert.assertNotNull(connection);
  Statement statement1 = connection.createStatement();
  Assert.assertNotNull(statement1);
  Statement statement2 = connection.createStatement();
  Assert.assertNotNull(statement2);
  connection.close();
  Assert.assertTrue(statement1.isClosed());
  Assert.assertTrue(statement2.isClosed());
}
```

Fig. 4. A failing test for the HikariCP bug whose fix is shown in Figure 3

From the above two examples, we can notice how the *future test cases* can impact the patches regenerated by APR. In practice, if APR is ever used to repair bugs, the existing fault-revealing test case must exist. The future test cases would typically be unavailable since APR has to fix the bugs before developers fix them and create such tests to confirm the absence of the bugs. Thus, evaluating APR on datasets containing fault-revealing tests whose construction involve *future test case* of bug fixes may not fully reflect how APR would perform in reality.

Since most datasets available for APR evaluation today include fault-revealing tests that belong to the future, there is a need for a new dataset. In this work, we create a dataset corresponding to bug fixes of CI failures. Fault-revealing tests in CI failures are created before bug fixes are submitted, and thus, assume no future test cases – developers do not know in advance how and where to fix the bugs at the time test cases are written; indeed, the bugs and even the buggy code may not have existed at the time the test cases are written.

III. Benchmark Construction Methodology

In this section, we review the criteria for constructing our defect benchmark. Then, we describe the steps to create the benchmark by following the criteria. Figure 5 depicts an overview of the process we follow to construct our benchmark.

A. Benchmark Criteria

Our constructed benchmark needs to possess the following desired criteria:

1) *Regression Bugs from CI* – wherein no prior knowledge on bug fixes is involved in constructing fault-revealing tests.
2) *Diversity* – wherein bugs are from a diverse set of real-world large software systems and well-categorized into various categories.

To satisfy *future test cases* criterion, we focus on continuous integration (CI) failures, in which fault-revealing tests are created before bug fixes are constructed by developers. For *diversity*, we iterate through over thousands of publicly available projects on GitHub – a popular code hosting service.

To collect real bugs from CI failures, we focus on GitHub projects that use CI – a widely-used automated build and testing methodology in modern software development. The property of CI that enables us to collect bugs is its constant monitoring and testing of code changes. Particularly, every time a code change is submitted to the version control system of a CI-employed project, the new code version of the project is automatically built and tested against the existing test suite by CI. A failed build containing failing tests indicates bugs.

[6] HikariCP (https://github.com/brettwooldridge/HikariCP/) is a popular high-performance JDBC connection pool. Its GitHub repository received more than 6,000 stars and has been forked close to 1,000 times.

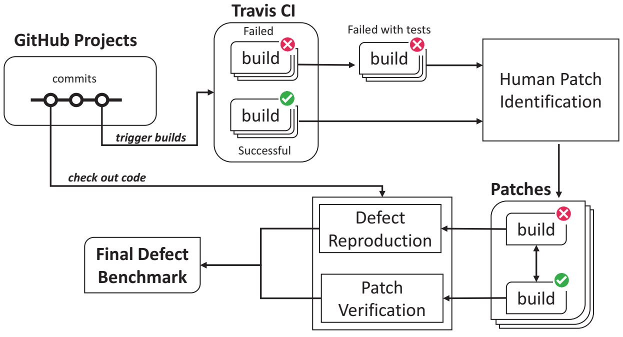

Fig. 5. The process of benchmark construction.

Note that each *build* produced by CI usually contains the following information: the source code snapshot used (e.g., a reference to a Git commit), the result of the build (e.g., success or fail), and the build log trace. This information allows us to identify project's versions wherein bugs via CI failures occur.

B. Phase 1: Project Selection

We focus on GitHub projects that employ Travis CI [2] given its popularity among several CI services available. Travis CI is a state-of-the-art CI service which is well integrated into the GitHub infrastructure. It also provides APIs to obtain CI build information. Obtaining build information from various Travis CI projects by reproducing builds, however, is a time-consuming process. Fortunately, this build information is made available via TravisTorrent [5], which allows us to access a database of hundreds of thousands of analyzed Travis CI builds in the matter of seconds. The projects in TravisTorrent are non-forked, sufficiently popular (> 10 watchers on GitHub) and have a history of Travis CI use (> 50 builds) [5]. We use the TravisTorrent database snapshot of 2017/02/08, which contains 2,022 projects.

We follow these steps to select projects:

1) We retain projects that use Java as the main programming language. We obtain 316 Java projects from TravisTorrent.
2) Using the number of commits that a project has as a proxy for measuring its activity, we filter projects whose number of commits is fewer than 1,000. After this step, there are 177 projects left.
3) We remove projects that cannot be easily configured among the remaining projects. First, we only retain projects that use Maven – an automated build tool that can help easily build and test projects. For example, we can use the command `mvn install` to build the project and the command `mvn test` to run the tests of the project. Second, we remove projects related to mobile or distributed platforms, since they may require various additional resources for building and running tests. For example, Android projects require emulators, which rely on different Android SDK frameworks. Finally, we get 91 projects.

Figure 6 presents the number of builds in the 91 projects. The mean and standard deviation of builds of these projects are 2,791 and 3,067, respectively.

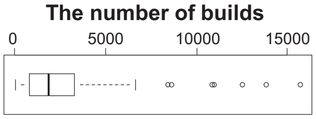

Fig. 6. Project builds.

TABLE III
PROJECTS HAVING MOST PATCHES AT THE END OF PHASE 2.

Project	Failed builds	Builds with test failures	PR builds with test failures	#Patches
apache/pdfbox	2,018	876	5	96
biojava/biojava	405	397	160	23
brettwooldridge/HikariCP	338	233	76	20
nutzam/nutz	552	316	27	17
rackerlabs/blueflood	558	294	205	12
structr/structr	843	406	4	12
datacleaner/	627	418	132	10
geoserver/geoserver	5,499	1,456	652	9

C. Phase 2: Defect and Patch Selection

In this step, we collect bug fixes along with their associated buggy and correct versions from the 91 projects obtained in the previous step. To do this, we iterate through the development history of the projects, and collect failed builds which are triggered by test failures to identify buggy versions and failing test cases. We then search for successful builds that fix those failures to identify corresponding correct versions of the buggy ones. By this way, we can obtain ground-truth patches – the changes made in correct versions as compared to buggy ones, and fault revealing test cases. We further illustrate the details of our defect and patch selection process step-by-step as follow.

First, we identify failed builds by first collecting details of metadata on each build using APIs provided by Travis CI. We then analyze the build logs to identify the builds with test case failures. We are not interested on failed builds caused by compilation errors, configuration errors, etc as our focus is only on the bugs from source code only. To find builds with test case failures, we use a regular expression to check whether a build log contains keywords such as "failed tests" or "tests in error", since we observe that the logs of failed builds often contain exception stack traces, which include these specific keywords.

Second, for each failed build with test failures as identified above, we search for the associated successful build, which contains the commit change that fixes the failure. There are two different kinds of commits that can trigger a build for GitHub projects using CI, i.e., commits *pushed* by the projects' members and commits in *pull requests* proposed by external developers of the projects. For failed builds triggered by *pushed* commits, we find the next successful build triggered by *pushed* commits with the same developer in build history. It is more likely that the next commit fixes the latest fault of a developer because CI can give timely feedback to let developers know whether their commit changes break the system or not.

For failed builds triggered by commits in *pull requests*, we find the next successful build with the same pull request ID.

Notice that sometimes we cannot find the next successful build since the pull request is not merged into master branch.

By comparing the two commits in the failed build and successful build using GitHub API, we can obtain changes representing a bug fix (i.e., a patch). However, GitHub might return no patch if the changed files are totally different. One could proceed to find another successful build submitted at a later time to obtain a patch, but the further away a successful build is from the failed build, the more likely the changes will be noisy. We thus ignore the failed build in this case and proceed to the next failed build. There are 9,404 patches found at the end of this step.

Finally, we further filter the collected patches since not all patches satisfy our criteria mentioned in Section III-B. Since we only focus on Java and most APR tools cannot fix bugs in code written in multiple languages at the same time, we remove patches that include changes to non-Java source code files, e.g., some changes are made on configuration files in text or XML format. Finally, we remove the patches that contain changes on test cases, because we want to focus on bugs on source code rather than test code. At the end of this step, we get 399 patches from 67 projects.

Table III presents the projects with the most patches at the end of this phase. For each project, we also show the number of failed builds, builds with test failures, and PR builds with test failures in this table. PR builds with test failures is a subset of builds with test failures.

D. Phase 3: Local Defect Reproduction and Patch Verification

After obtaining the defects, fault revealing test cases, and their corresponding patches, we reproduce these defects in our local machine and verify whether the corresponding patches can really fix the defects. To do this, we follow the subsequent steps one-by-one.

Checking out the code for both buggy and fixed versions. To get exactly the same source code of both versions, we clone the project repository from GitHub using the commit identifier in the builds. However, checking out code could fail in the following cases: 1) commits might have been deleted due to amended commits or updated branches performed with the push force option; 2) the branch that contained the build's commit was deleted if the type of build is *pull request*. In such cases, we stop the defect reproduction phase and remove the defects from our dataset.

Building project and running tests for buggy versions. This step is to check whether we can reproduce defects in the buggy version of each defect. Generally, we use the command `mvn install` to build the project, which usually includes resolving the dependencies, compiling source code, running test cases, and packaging the project. For each build, we create a new local cache directory for Maven to avoid conflicts among the same dependent libraries. To execute the tests of a project, we run the command `mvn test`.

Then, for each buggy version, we check the local test outputs to confirm that there are failed test cases. If so, we compare the failed test cases with the outputs in build logs generated by CI. If both the failed test cases and the thrown exceptions are the same, the defect is considered locally reproducible. Otherwise, the defect is discarded.

Building project and running tests for fixed versions. This step is to verify whether the commit change in the fixed versions really fixes the defects in the corresponding buggy versions. We build and run tests for the fixed versions in the same way as the buggy versions. Then, if there are no failed test cases, the human patch is considered to really fix the bug; otherwise, we discard the patch. At the end of this step, we get 102 defects from 40 projects. Filtering the bugs took us approximately 900 man hours, including tasks such as manually building and resolving build issues, e.g., missing dependencies, and manually reproducing the bugs, etc. We spent two or three hours resolving build issues, beyond which we left the bug to inspect later at low priority. We acknowledge that there is a trade-off to be made here between the accuracy and inspection time.

IV. Data Statistics

Our benchmark contains 102 defects from 40 systems obtained via several processing steps described in Section III. To further justify the quality of our constructed benchmark, in this section, we summarize project included in the benchmark, and then categorize defects into several classes.

A. Constituent Projects

Table IV summarizes the projects in our defect benchmark. Due to space limitation, we only present the details of 10 projects that have at least three defects in the benchmark. We note that the 40 projects in the benchmark are diverse in terms of types and sizes. The projects span JDBC connection pool, GPS tracking system, library for interaction with Bitcoin exchanges, JavaScript compiler, style and grammar checker, and many more. As Table IV shows, the project that has the largest number of lines of code (LOC) is `hapi-fhir` with ∼1.8 millions LOC, while the project that has the smallest number of LOC is `HikariCP` with ∼12 thousands LOC. The projects in the benchmark are developed by both large and small teams; the mean and standard deviation of the number of project contributors are 86 and 72, respectively. We also compute the number of test cases and their corresponding mean LOC for each project. The mean and standard deviation of test case count are 521 and 633, respectively. Among these projects, `fastjson` has much larger number of test cases than others despite having not very large LOC. Moreover, we collect statement coverage for each project using a Maven plugin – Cobertura [1]. The mean and standard deviation of statement coverage are 54% and 30%, respectively.

B. Defect Categorization

We further justify the diversity of our benchmark by characterizing the constituent defects. We divide defects in the benchmark into several different categories, of which some are obtained from the Java defect classes proposed by Pan et

TABLE IV
SOFTWARE PROJECTS IN OUR BENCHMARK.

				test cases		
Project	#Defects	kLOC	#Contributors	Count	LOC (mean)	description
brettwooldridge/HikariCP	9	12	78	47	145	high-performance JDBC connection pool
datacleaner/DataCleaner	8	131	34	392	68	data quality solution
alibaba/fastjson	7	149	69	2,375	48	JSON parser/generator
jamesagnew/hapi-fhir	5	1,861	59	743	222	API for HL7 FHIR
owlcs/owlapi	4	154	16	264	126	API for W3C Web Ontology Language
tananaev/traccar	4	49	73	251	32	GPS tracking system
nutzam/nutz	3	27	50	187	76	web framework
vavr-io/vavr	3	128	72	340	100	language extension for Java 8
apache/pdfbox	3	91	74	489	53	library for PDF documents
caelum/vraptor4	3	92	53	487	51	web MVC framework
google/closure-compiler	3	72	23	191	229	JavaScript checker and optimizer
other projects (mean)		165	94	544	97	
all (mean)		199	86	521	112	

al. [45] and Tan et al. [53]. Each defect category is characterized by the changes made to fix the defect. Table V presents the defect categories for our dataset and the corresponding number of defects belonging to each category. There are six categories of defects in our dataset, i.e., *if condition*, *method call*, *variable*, *assert*, *exception*, and *others*. We describe the detail of each category as follows:

If condition: The defects in this category are related to condition check in *if* statements. This category has the most number of defects (26/102) in our dataset. Out of these 26 defects, 7 and 1 defects are fixed by adding additional expressions to tighten (i.e., with && operation) and loosen (i.e., with || operation) condition check in *if* statements, respectively. 11 defects are fixed by only modifying the original condition expressions in *if* statements. 7 defects are fixed by inserting an *if* statement to ensure that a precondition is met. We have two sub categories for this case: one is to add an *if* statement that is copied from elsewhere in the project source code, another is to add an *if* statement that might be created by developers from scratch. Both of the categories have five defects in our dataset. The remaining two defects are fixed by adding an *if-else* code block.

Method call: There are 21 defects in this category, which are fixed by applying changes on method calls. The number of defects that can be fixed by *adding/removing* one or more method calls are 3 and 2 defects, respectively. Two defects are fixed by replacing a method call. The new method call usually has a similar name or function to the old one. The sub category *modify method invoker* is a defect fix pattern that replaces the object from which a method is called. The new object must be of the same type as the old one. The other two defect fix patterns are *change parameter list* (3 defects) and *change parameter value* (7 defects). The first case changes a method call by using different numbers of parameters, or different parameter types. The second case changes the expression passed into one or more parameters of method calls.

Variable: There are 24 defects in this category, which are fixed by applying changes on variables. Out of these 24 defects, 6, 5, and 3 defects are fixed by *adding*, *modifying*, and *removing* one or more statements of variable assignment, respectively. There is 1 defect fixed by removing some keywords ("final", "static", "transient" in our dataset) occurring before some variables. The remaining 9 defects are fixed by replacing the type of a variable with another one.

Assert: The defects in this category are triggered when developers change a certain part of code but do not modify the corresponding *assert* check. There are only two defects that belong to this category, which are fixed by modifying and removing one *assert* statement, respectively.

Exception: The defects in this category are fixed by preventing certain exceptions to interrupt the program execution. We have six cases: one adds a try catch block and the other five changes the exception type.

Others: There are still 24 defects for which we cannot determine an obvious category. Out of these 24 defects, three defects are fixed by adding an interface ("Serializable" in both cases); one defect is fixed by adding return and 11 defects by modifying the *return* statement, respectively; two defects is fixed by modifying the resource files and seven bugs are due to changes in the method implementation.

V. EMPIRICAL EVALUATION

In this section, we seek to answer two research questions, demonstrating various features and the usefulness of our dataset presented in Section 4.

A. Characteristics of Failure-exposing Test Cases

RQ1: How different are failure-exposing test cases in our dataset than those included in Defects4J?

Methodology. To answer this research question, we compute a number of statistics shown in Table VI from failure-exposing test cases of 102 bugs in our dataset and compare with those of 395 bugs from Defects4J [14]. In total, we compute 4 statistics (i.e., S1, S2, S3, and S4) listed in Table 4 to assess *quantity*, *bug localizability*, and *similarity* of failure-exposing test cases to bug fix patches. In particular, S1 specifies the average number of failure-exposing test cases in each bug. The more failure-exposing test cases, the larger is the region

TABLE V
DEFECT CATEGORIES.

Category	Sub Category	#Defect
if condition	tighten if condition	7
	loosen if condition	1
	modify if condition	11
	insert if condition from code	2
	insert if condition from unknown	3
	insert if condition with else	2
method call	add method call	3
	remove method call	2
	replace method call	2
	modify method invoker	4
	change parameter list	3
	change parameter value	7
variable	add variable assignment	6
	modify variable assignment	5
	remove variable assignment	3
	remove keyword before variable	1
	replace variable declaration type	9
assert	modify assert expression	1
	remove assert expression	1
exception	add try-catch block	1
	modify exception type	5
others	move statement	0
	add interface	3
	insert return	1
	change return	11
	replace array access	0
	remove annotation	0
	modify resource files	2
	modify method implementation	7

TABLE VI
STATISTICS EXTRACTED FROM FAILURE-EXPOSING TEST CASES AND SOURCE CODE OF EACH BUG.

ID	Description
	Notations
RC	Number of unique classes that have to be repaired and explicitly called by failure-exposing test cases
C	Number of unique classes explicitly called by failure test cases
RM	Number of unique methods that have to be repaired and explicitly called by failure-exposing test cases
M	Number of unique methods explicitly called by failure test cases
	Statistics
S1	Average number of failure-exposing test cases
S2	Average of RC / C
S3	Average of RM / M
S4	Average number of shared identifiers between failure-exposing test cases and patches

of code that may be deemed suspicious, and thus enlarging the search space for repair. S2 and S3 are the percentage of classes and methods that need to be repaired among those are explicitly called by test-exposing test cases – they measure the bug localizability potential of the test cases. The higher values of S2 and S3 are, the higher chances failure-exposing test cases contain hints leading to root causes of bugs. Finally, S4 measures similarity between failure-exposing test cases and bug fix patches in terms of shared identifiers.

Results. All Projects: We note that mean of S1 for our dataset and Defects4J are 15.39 and 2.37, respectively. This shows that the average number of failure-exposing test cases per bug in our dataset is more than *six times* larger than the number for Defects4J. Wilcoxon rank-sum test [60] shows that the difference is statistically significant (i.e., p-value=0.000016).

TABLE VII
APR TOOLS EVALUATION FOR DEFECTS4J AND OUR DATASET. JGEN AND CARD STAND FOR JGENPROG AND CARDUMEN RESPECTIVELY.

Defects4J - 224 Bugs

Tool	**jGen**	**jKali**	**Arja**	**RsRepair**	**Mean**
Patched Bugs	27	22	59	44	38
Patched %	12.05%	9.82%	26.34%	19.64%	16.96%

Our dataset - 102 bugs

Tool	**jGen**	**jKali**	**Arja**	**RsRepair**	**Card**	**Mean**
Patched Bugs	4	4	0	0	3	2.5
Patched %	3.92%	3.92%	0%	0%	2.94	2.16%

The mean of S2 for our dataset and Defects4J are 0.042 and 0.142, respectively; mean of S3 in our dataset and Defects4J are 0.026 and 0.088, respectively. We note that failure-exposing test cases in our dataset are approximately *three times* less localizable than those in Defect4J. In other words, Defects4J's failure-exposing test cases have much higher chance leading to root causes of bugs than those in our dataset. Wilcoxon rank-sum test shows that the differences in S2 and S3 between our dataset and Defect4J are statistically significant with p-values of 0.000620 and 0.002736, respectively.

The mean of S4 for our dataset and Defects4J are 0.426 and 0.604, respectively. This indicates that failure-exposing test cases in our dataset is less similar to bug fix patches than those in Defects4J by approximately 70%.

> Failure-exposing test cases per bug in our dataset are more than six times larger than those for Defects4J; they are also three times less localizable and close to 70% less similar to bug fix patches.

B. Effectiveness of APR

RQ2: Are there differences in automated program repair effectiveness evaluated using our dataset as compared to Defects4J?

Methodology. Using our dataset, we evaluate five well-known APR techniques namely GenProg, Kali, Cardumen implemented in Astor [39], Arja [65],and RsRepair [48]. We subsequently compare the effectiveness of APR evaluated on our dataset with that on Defects4J that is recently reported by Martinez et al. and Chen et al. [8], [38]. We note that the effectiveness of APR tools on a dataset represents the overall difficulty of the dataset on repair task. Thus, our evaluation metric to estimate an average score for program repair effectiveness is *correct patch rate* measured as $\#CP/(\#Tools*\#Bugs)$, where $\#CP$ is the total number of correct patches generated by all the APR tools used, $\#Tools$ is the number of APR tools used to generate repairs, and $\#Bugs$ is the number of bugs used for evaluating APR in total. Note that a machine-generated patch is judged as correct if: (1) it passes all tests in existing test suite in fixed commit, and (2) it is identified as semantically equivalent to the ground-truth patch by manual human evaluation; each machine-generated patch is manually reviewed by two authors of this paper, wherein each author provides their own label for the patch (i.e., correct, incorrect, or unknown) independently, and reciprocally discusses on disagreement cases until a consensus is achieved. Semantic equivalence of patches is evaluated by following rules described by Liu et al. [32]. For example, one of the 10 rules described in Table 3 in [32] is "Unnecessary code uncleaned"; this rule means that two patches are considered semantically equivalent even if applying one of them result in harmless unnecessary code that is uncleaned.

Results. Experimental results show that, the five APR tools that were run against the 102 bugs, found patches for only 6 bugs in common. Among the 6 common bugs patched, jKali, jGenProg and Cardumen patched 4, 4 and 3 bugs respectively. Based on the study in [65] and our experiments, Table VII presents the mean patch rate of APR tools against Defects4J and our benchmark which is 16.9% and 2.16% respectively. This indicate 8 times lower patch rate on our benchmark despite the fact that our dataset has fewer lines of code modified per bug on average as compared to Defects4J as shown in Table II.

> Overall, the results indicate a lower patch rate on our dataset as compared to Defects4J, suggesting that it is more difficult for APR to generate correct patches on our dataset.

VI. Discussion

Beyond Program Repair. Our constructed dataset can be useful to evaluate automated techniques in other research areas beyond program repair that use dynamic analysis. One such area is fault localization [3], whose research interests have intensified in recent years [4], [46], [47], [66].

Fault localization techniques (FL) have been evaluated on several benchmarks in various studies [4], [46], [47], [66], which either contain artificial faults manually intentionally injected by human, e.g., SIR [10], or involve bugs with *future test cases*, e.g., Defects4J [14]. Our dataset can serve as a means to estimate how well FL would perform in practice. It is possible that the evaluation results of FL on our dataset would be different from that on other less realistic benchmarks as shown in previous studies [4], [66].

We believe that by focusing on realistic evaluations of APR, such as evaluations on continuous integration as ours, would help bring APR closer to real-world adoption in the future as it helps evaluate APR more fairly to reflect the true performance of APR in practice. Also, more efforts are needed from the APR community to help cultivate datasets that facilitate such evaluations. The take home messages of our paper for APR researchers include: (1) please consider the benchmark that we have created in the evaluation of future APR tools (as the future should not be used to fix the past), (2) please create larger benchmarks considering curated continuous integration failures from various systems.

Threat to Validity. Threats to *internal validity* relate to errors in our implementation and experiments. We have rechecked our implementation and experiments and fixed errors that we have found. Still, there could be additional errors that we did not notice. Additionally, one potential threat that can affect the quality of our dataset is the reproducibility of defects we collected. We have carefully reproduced defects as described in Section III-B and discarded ones that are not reproducible.

Threats to *external validity* correspond to the generalizability of our findings. In this study, we have collected 102 real bugs from 40 different Java programs, and evaluated APR tools on the dataset. Still, more programs with more real bugs can be collected, and more APR tools can be evaluated to mitigate the threats further. We plan to do this in our future work.

Threats to *construct validity* correspond to the suitability of our evaluation metrics. For RQ1, we use some intuitive metrics to characterize failure-exposing test cases. For RQ2, we use correct patch rate to measure the effectiveness of APR tools, which has been similarly used in prior studies, e.g., [23], [25]. Still, there could other metrics that may be used for the two RQs, we leave that investigation for future work.

VII. Related Work

In this section, we highlight related work on automated program repair including state-of-the-art techniques, benchmarks and empirical studies.

A. Automated Program Repair

Weimer et al. propose AE that leverages an adaptive search strategy to find similar syntactic repairs [59]. Qi et al. propose RSRepair by employing a random search strategy to determine repair candidates, and RSRepair is shown to be more effective than GenProg on a subset of GenProg's benchmark [49]. Long et al. propose SPR that combines staged program repair and condition synthesis to effectively search for repair candidates [34]. Prophet is a novel approach that infers probabilistic models for assigning probabilities to repair candidates in the search space [36]. Le et al. propose HDRepair that leverages history of bugs fixes of thousands of projects from GitHub to guide the repair process [24]. It uses genetic programming to generate repair candidates, and rank the candidates based on the likelihood of being correct, which is measured by how frequent the changes made by the candidates appear in the bug fix history.

Konighofer and Bloem utilize symbolic execution to construct repair candidates for linear expressions [17]. Nguyen et al. propose a constraint-based approach, named SemFix, that leverages symbolic execution, constraint solving, and program synthesis for automated program repair [42]. There are other existing techniques that employs abstract interpretation, unguided by test suites, but these techniques require specially-written, well-specified code (e.g., [33]). Ke et al. introduce SearchRepair that leverage semantic code search by encoding human-written code portions as SMT constraints on input-output behavior [15]. Angelix is a semantics-based method that introduces a novel lightweight repair constraint for repairing large-scale real-world software systems [41]. Le et al. translate and extend Angelix [41] to work on Java programs [19]. Le et al. subsequently also proposed regression errors repair for Java program [20].

B. Program Repair Benchmarks

IntroClass benchmark contains several hundreds of small student-written C programs [29]. Tan et al. [54] create Codeflaws that contains 3,902 defects from 7,436 small programs from programming contests hosted on Codeforces[7]. Similar to IntroClass, each program in Codeflaws contains two independent test suites, in which one test suite can be used to validate APR-generated patches [54]. Nilizadeh et al. provided a dataset of bugs equipped with formal specifications [43], [44].

GrowingBugs [13] is a recent bug repository composed of 1,381 real-world bugs and their concise patches, automatically collected from 151 well-known and widely used Java applications. Different from GrowingBugs, our dataset is specifically for regression bugs.

BugSwarm [56] a collection of the unprocessed bugs from CI failed builds, it contains 3091 pairs of failing and passing continuous integration builds. The dataset is created by means of automated mining of fail-pass pairs from CI builds [56]. There has been critical analysis shows several limitations in the bugswarm benchmark: only 50 Java and 62 Python bugs out of 3091 (pair of builds) are suitable to evaluate techniques for automatic fault localization or program repair. This result has been obtained by applying the seven filters (1) Test-case failure, (2) Only change source file, (3) No test changed, (4) Build reproduced five times, (5) Available Docker image, (6) Unique commit, (7) Not Empty diff [11].

A recent study [31] showed that 90% of benchmark bugs in Defects4J [14], Bugs.jar [51] and Bears [37] are associated with bug-triggering test cases that have been processed, i.e., added/updated after the bug is reported. We also note that the benchmark authors have taken steps to curate the buggy programs to facilitate program repair tools. Several APR systems may be overfitting to the available benchmarks, therefore lacking generalizability on other bug targets.

Our work introduces a new bench mark, which includes buggy versions, correct versions, test suites, and all of them available in our GitHub repository CIBugs[8]. Different from existing benchmarks, bugs included in our benchmark are all from continuous integration (CI) failures that are detected using test cases that were created *before* the bugs were detected.

C. Empirical Studies on Program Repair

Qi et al. manually evaluate patches produced by previous heuristic techniques to highlight the risks that test cases pose when guiding repair search [50]. Smith et al. empirically and systematically evaluate the overfitting issue in search-based program repair techniques [52], including GenProg [58] and RSRepair [49]. Long et al. show that search spaces of search-based approaches are often large and correct repair candidates sparsely occur within the search spaces [35]. Le et al. assess the effectiveness of synthesis engines when deployed for semantics-based program repair [23]. Recently, Le et al. proposed a new approach to automatically assess patch correctness via program invariants and machine learning [27].

Le Goues et al. quantitatively assess impact of generated patches in a closed-loop system for detection and repair of security vulnerabilities [30]. Kim et al. assess relative acceptability of patches generated by a novel technique via a human study [16]. Fry et al. conduct a human study of patch maintainability, finding that generated patches can often be as maintainable as human patches [12].

Recently, Urli et al. report their experiences and insights gained for designing a program repair bot namely Repairnator [57]. It employs three APR tools namely Nopol [63], Astor [39], and NPEFix [9], to operate on 11523 test failures over 1609 open source projects. Repairnator focuses on highlighting challenges and experiences of building a repair bot, while our work focuses on building benchmarks for APR tools. Repairnator's experiment data, despite being released, does not possess suitable features that are amenable for future APR evaluations. In particular, Repairnator's data is raw, does not provide ground-truth fixes of bugs, and is not well-bundled to support controlled experiments like our dataset.

Our work complements the above-mentioned work by investigating another angle, namely, impact of *future test cases* on effectiveness of APR solutions. We highlight that test suites that are created *before* patches were made have different properties than those that are created *at the same time* or *after* patches were submitted. We provide some empirical evidence that test suites that satisfy the *no future test case* criterion better reflects reality and poses a harder challenge that is yet to be solved by the APR research community.

VIII. Conclusion and Future Work

In this paper, we argued that benchmarks that have been used to evaluate the effectiveness of APR, e.g., Defects4J [14] and ManyBugs [29] still may not fully reflect how well APR would perform in practice. We constructed a benchmark, containing 102 bugs identified via continous integration failures from 40 real-world large programs to evaluate APR. We find that failure-exposing test cases in our dataset are substantially more, less localizable and less similar to bug fix patches. We subsequently applied our benchmark to evaluate five well-known APR tools namely GenProg, Kali, Cardumen, Arja and RsRepair. Experiment results suggest that it is six time more difficult for APR tools to generate correct patches on our dataset as compared to existing dataset, i.e., Defects4J.

As future work, we plan add more bugs from more programs and evaluate more APR techniques on our dataset. We also plan to use our dataset to evaluate other techniques that leverage dynamic analysis, such as fault localization, etc.

[7] http://codeforces.com/

[8] https://github.com/CI-Bugs/Repo1

IX. Acknowledgments

This research was funded (partially) by the Australian Government through the Australian Research Council's Discovery Early Career Researcher Award, project number DE220101057. This research / project is supported by the National Research Foundation, Singapore, under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

[1] Cobertura. http://cobertura.github.io/cobertura/.

[2] Travis ci. https://travis-ci.org/.

[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98, 2007.

[4] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 177–188. ACM, 2016.

[5] Moritz Beller, Georgios Gousios, and Andy Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Proceedings of the 14th working conference on mining software repositories*, 2017.

[6] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. Technical report, University of Cambridge, Judge Business School, 2013.

[7] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *International Conference on Software Engineering*, ICSE'11, pages 121–130, 2011.

[8] Liushan Chen, Yu Pei, and Carlo A Furia. Contract-based program repair without the contracts. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, pages 637–647. IEEE, 2017.

[9] Benoit Cornu, Thomas Durieux, Lionel Seinturier, and Martin Monperrus. Npefix: Automatic runtime repair of null pointer exceptions in java. *arXiv preprint arXiv:1512.07423*, 2015.

[10] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[11] Thomas Durieux and Rui Abreu. Critical review of bugswarm for fault localization and program repair, 2019.

[12] Zachary P. Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 177–187, 2012.

[13] Yanjie Jiang, Hui Liu, Xiaoqing Luo, Zhihao Zhu, Xiaye Chi, Nan Niu, Yuxia Zhang, Yamin Hu, Pan Bian, and Lu Zhang. Bugbuilder: An automated approach to building bug repository. *IEEE Transactions on Software Engineering*, pages 1–1, 2022.

[14] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis*, ISSTA '14, pages 437–440, 2014.

[15] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *International Conference on Automated Software Engineering (ASE)*, pages 295–306, 2015.

[16] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering*, ICSE '13, pages 802–811, 2013.

[17] Robert Könighofer and Roderick Bloem. Automated error localization and correction for imperative programs. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 91–100. FMCAD Inc, 2011.

[18] Dinh Xuan Bach LE. Overfitting in automated program repair: Challenges and solutions. 2018.

[19] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. Jfix: Semantics-based repair of java programs via symbolic pathfinder. In *International Symposium on Software Testing and Analysis*, ISSTA'17, 2017 (to appear).

[20] Xuan-Bach D Le and Quang Loc Le. Refixar: Multi-version reasoning for automated repair of regression errors. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 162–172. IEEE, 2021.

[21] Xuan Bach D. Le, Quang Loc Le, David Lo, and Claire Le Goues. Enhancing automated program repair with deductive verification. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 428–432, 2016.

[22] Xuan-Bach D Le, Tien-Duy B Le, and David Lo. Should fixing these failures be delegated to automated program repair? In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 427–437, 2015.

[23] Xuan-Bach D Le, David Lo, and Claire Le Goues. Empirical study on synthesis engines for semantics-based program repair. In *International Conference on Software Maintenance and Evolution*, ICSME'16, pages 423–427, 2016.

[24] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 213–224. IEEE, 2016.

[25] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. 2017.

[26] Xuan Bach Dinh Le, Duc Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax-and semantic-guided repair synthesis via programming by example. *FSE. ACM*, 2017.

[27] Thanh Le-Cong, Duc-Minh Luong, Xuan Bach D Le, David Lo, Nhat-Hoa Tran, Bui Quang-Huy, and Quyet-Thang Huynh. Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning. *IEEE Transactions on Software Engineering*, 2023.

[28] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, ICSE'12, pages 3–13, 2012.

[29] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *Transactions on Software Engineering (TSE)*, 41(12):1236–1256, Dec. 2015.

[30] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[31] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, 171:110817, 2021.

[32] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 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*, pages 615–627, 2020.

[33] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. *SIGPLAN Not.*, 47(10):133–146, October 2012.

[34] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 166–178, 2015.

[35] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *International Conference on Software Engineering (ICSE)*, pages 702–713. ACM, 2016.

[36] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Symposium on Principles of Programming Languages (POPL)*, pages 298–312, 2016.

[37] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–478. IEEE, 2019.

[38] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, pages 1–29, 2016.

[39] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444. ACM, 2016.

[40] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *International Conference on Software Engineering (ICSE)*, pages 448–458. IEEE Press, 2015.

[41] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE)*, pages 691–701. IEEE, 2016.

[42] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE Press, 2013.

[43] Amirfarhad Nilizadeh, Marlon Calvo, Gary T Leavens, and Xuan-Bach D Le. More reliable test suites for dynamic apr by using counterexamples. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 208–219. IEEE, 2021.

[44] Amirfarhad Nilizadeh, Gary T Leavens, Xuan-Bach D Le, Corina S Păsăreanu, and David R Cok. Exploring true test overfitting in dynamic automated program repair using formal methods. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 229–240. IEEE, 2021.

[45] Kai Pan, Sunghun Kim, and E James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[46] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620. IEEE, 2017.

[47] Alexandre Perez, Rui Abreu, and Arie van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *Proceedings of the 39th International Conference on Software Engineering*, pages 654–664. IEEE Press, 2017.

[48] Yuhua Qi, X. Mao, Y. Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[49] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

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

[51] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 10–13, 2018.

[52] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543. ACM, 2015.

[53] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 180–182. IEEE Press, 2017.

[54] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. Codeflaws: A programming competition benchmark for evaluating automated program repair tools. In *ICSE Poster*, 2017. To appear.

[55] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *Planning Report, NIST*, 2002.

[56] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 339–349, 2019.

[57] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. How to design a program repair bot? insights from the repairnator project. In *40th International Conference on Software Engineering, Track Software Engineering in Practice (SEIP)*, pages 1–10, 2018.

[58] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, 2010.

[59] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 356–366. IEEE Press, 2013.

[60] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83, 1945.

[61] Qi Xin and Steven P Reiss. Identifying test-suite-overfitted patches through test case generation. In *International Symposium on Software Testing and Analysis*, pages 226–236. ACM, 2017.

[62] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *International Conference on Software Engineering*, pages 416–426. IEEE Press, 2017.

[63] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *Transactions on Software Engineering*, 2016.

[64] Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, pages 740–751. ACM, 2017.

[65] Yuan Yuan and Wolfgang Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *CoRR*, 2017.

[66] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. Boosting spectrum-based fault localization using pagerank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 261–272. ACM, 2017.