P3: A Dataset of Partial Program Patches

Dirk Beyer
LMU Munich
Munich, Germany
dirk.beyer@sosy.ifi.lmu.de

Lars Grunske
Humboldt-Universität zu Berlin
Berlin, Germany
grunske@informatik.hu-berlin.de

Matthias Kettl
LMU Munich
Munich, Germany
matthias.kettl@sosy.ifi.lmu.de

Marian Lingsch-Rosenfeld
LMU Munich
Munich, Germany
marian.lingsch@sosy.ifi.lmu.de

Moeketsi Raselimo
Humboldt-Universität zu Berlin
Berlin, Germany
raselimm@informatik.hu-berlin.de

ABSTRACT

Identifying and fixing bugs in programs remains a challenge and is one of the most time-consuming tasks in software development. But even after a bug is identified, and a fix has been proposed by a developer or tool, it is not uncommon that the fix is incomplete and does not cover all possible inputs that trigger the bug. This can happen quite often and leads to re-opened issues and inefficiencies. In this paper, we introduce P3, a curated dataset composed of incomplete fixes. Each entry in the set contains a series of commits fixing the same underlying issue, where multiple of the intermediate commits are incomplete fixes. These are sourced from real-world open-source C projects. The selection process involves both automated and manual stages. Initially, we employ heuristics to identify potential partial fixes from repositories, subsequently we validate them through meticulous manual inspection. This process ensures the accuracy and reliability of our curated dataset. We envision that the dataset will support researchers while investigating partial fixes in more detail, allowing them to develop new techniques to detect and fix them. We make our dataset publicly available at https://gitlab.com/sosy-lab/research/data/partial-fix-dataset.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Partial Program Fixes, Supplementary Bug Fixes, Recurring Bugs

ACM Reference Format:
Dirk Beyer, Lars Grunske, Matthias Kettl, Marian Lingsch-Rosenfeld, and Moeketsi Raselimo. 2024. P3: A Dataset of Partial Program Patches. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3643991.3644889

1 INTRODUCTION

Debugging, which comprises fault localization, fault understanding, and fault repair, takes up a large part of resources of the software development process. It has been shown that fixes proposed by developers or automatically by a tool have a high likelihood of being incorrect [15, 24, 25]. Incorrect patches are prevalent, and typically manifest themselves as re-opened issues because an earlier attempt at the fix introduces another bug or misses some edge cases. Existing research has documented the extent to which these bad fixes exist: reports in these studies [2, 10, 18, 24] show that about 10 % – 33 % of bugs in several large-scale software projects required multiple fix attempts due to an initial incorrect patch.

We concentrate on the *bad-fix problem* [10]. Bad fixes fall into two categories: (*i*) bug fixes that introduce regressions and (*ii*) bug fixes that *partially* fix a bug and therefore only cover a fraction of bug inducing inputs. We focus on the second category of bad fixes here, and we call these *partial* or *incomplete patches.* Unlike in regression testing (first category), where many techniques, tools, and well-defined benchmarks to evaluate the bad fixes exist [13, 16], little attention has been paid to finding and fixing incomplete patches directly. While the existence of partial fixes has been studied before [2, 10, 12, 18, 24], insights from these studies are drawn from only a handful of programs and therefore do not readily lend themselves to evaluating new automated approaches to repair incomplete patches.

In this paper, we take a step forward in exploring the prevalence of partial patches on a large number of code repositories. More specifically, we present a dataset based on real-world C programs with real bugs that required multiple attempts to correctly (and completely) fix the bug. To assemble this dataset, we crawled 3 717 C repositories on GitHub and also added cases of incomplete patches from the Linux kernel. Identifying incomplete patches involves two steps: (*a*) *candidate selection*: we use heuristics based on events in each issue for crawled repositories and information in the commit messages of the Linux kernel; (*b*) *validation*: for each issue i, with related commits $X_1, \ldots, X_n$ ordered chronologically, we analyze if commits $X_2, \ldots, X_{n-1}$ are partial fixes of the problem occurring at X_1 with an expected fix X_n. Note that we ignore merge and intermediate commits, if they do not have any impact on the issue.

This paper builds on top of the ideas in our technical report [5]. The main extension to that paper is the validation and classification of 'true' partial fixes. We therefore introduce and release P3[1], a dataset comprising 187 programs with partial fixes. Most of them belong to control-flow bugs caused by missing or incorrect conditions in if-statements or loops. Other classes of partial fixes are memory-related, hardware-specific, or concurrency-related.

MSR '24, April 15–16, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0587-8/24/04
https://doi.org/10.1145/3643991.3644889

[1]P3 stands for Partial Program Patches

```
@@@-391,6 +391,8 @@int main(int argc, char **argv)
/* goto the last line that had a character on it */
for (; l->l_next; l = l->l_next)
  this_line++;
+ if (max_line == 0)
+ return EXIT_SUCCESS; /* no lines, so just exit */
```

Figure 1: First, partial fix to issue 422 with col

```
@@-396,7 +396,7 @@int main(int argc, char **argv)
/* goto the last line that had a character on it */
for (; l->l_next; l = l->l_next)
  this_line++;
- if (max_line == 0)
+ if (max_line == 0 && cur_col == 0)
  return EXIT_SUCCESS; /* no lines, so just exit */
```

Figure 2: Second, successful fix to issue 422 with col

Potential Applications. First of all, having a well-curated dataset enables a fair, accurate, and precise evaluation of tools for automatic program repair. Also, it supports proposals of new measures for the quality of patches. Second, the partial fixes in our dataset contain valuable information (e.g., bug location and program semantics) that can be exploited by specialized program-repair techniques, thereby making subsequent repair tasks easier. Since the location oracle is available by construction, these repair tools do not need any output from expensive and imprecise fault-localization methods and can instead solely focus on patch synthesis. Machine-learning-based program-analysis and repair techniques can also benefit from this dataset. New techniques in these areas can incorporate this set in their training pipelines, so they can better detect and predict incomplete patches and suggest ways on how to repair.

Example. We showcase a partial fix with the help of issue 422 [2] of the repository `util-linux`[3]. We derive our example from one of the utilities in the repository called `col`[4], which filters out the Unicode characters 'reverse line feed' (go up one line) and 'half-reverse line feed' (go up half a line) from a given input. In revision c6b0cb, `col` falsely printed a newline character for empty inputs. The changes in Fig. 1, trying to solve the problem by exiting without printing a newline if there are no line breaks in the input. However, it would still wrongly print a newline for input sequences with at least one character without line breaks. Therefore, an additional check was later added to the if-condition to fix the bug (cf. Fig. 2).

2 METHODOLOGY

Figure 3 summarizes the selection process. First, we automatically generate candidates from code repositories (⚙). Afterwards, we use heuristics to filter (🔍) the most promising candidates for manual inspection. The manual inspection is done by three authors (👤) in two phases. First, every author inspects a third of the candidates and passes 100 % of their candidates labeled 'partial fixes' (green arrow)

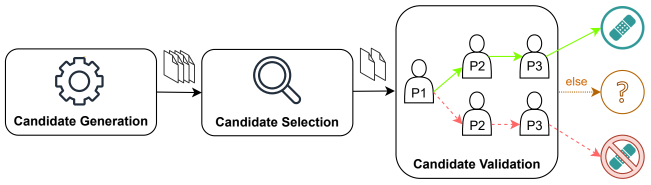

Figure 3: Selection process for the benchmark set

and 5 % of the candidates labeled 'no partial fix' (dotted red arrow) to the other authors. Second, the other authors cross-validate the candidates. We label a candidate as a 'partial fix' or 'no partial fix' if all authors agree on the classification. We label a candidate as 'unknown' (**?**) if it has not yet been inspected by all authors or if the authors do not agree on the classification. In the dataset, we keep all candidates, but add the above labels to the task-definition file (cf. Fig. 5).

2.1 Candidate Generation and Selection

In this section, we describe the heuristics employed to automatically obtain the most promising candidates for our dataset. Each candidate is a sequence of commits with some metadata related to them.

GitHub Issues. We query all C repositories on GitHub and sort them descending by the number of received stars. The number of stars is a good indicator for well-maintained and popular projects. We then go through all repositories and collect all closed issues. The issue together with a list of issue events are processed in two stages during the automatic candidate selection. First, we examine the issue events and check if the following criteria hold:

C_{reopen} The issue was reopened after it was closed and has at least one commit after reopening.

C_{status} The issue has multiple commits and the CI pipeline failed for at least one commit but not the last one.

Second, if at least one of these criteria holds, we consider the issue as a candidate for a partial fix. For each candidate, we store all available metadata from the issue's events. Issue events track discussions, comments, labels, actions, and commits. We track quantitative information such as the number of commits, the number of touched files, and changed lines. Additionally, we also track qualitative information such as the discussion on GitHub, the commit messages, helpful labels, and keywords. The metadata are later used to ease the manual validation.

Linux Kernel. The developers of the Linux kernel have the convention to tag the bug-inducing commit being fixed[5]. The reference between commits is done by adding "`Fixes:`" to the commit message[6]. We use this information to build sequences of commits by following the "`Fixes:`" references. If a sequence has at least three commits, then it is likely to contain a commit which introduced the bug, a partial fix, and a final solution. Therefore, we only consider sequences with more than three commits for the manual validation. All other sequences are discarded.

[2]https://github.com/util-linux/util-linux/issues/422
[3]https://github.com/util-linux/util-linux
[4]https://man7.org/linux/man-pages/man1/col.1.html

[5]https://lwn.net/Articles/914632/
[6]https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c511851de162

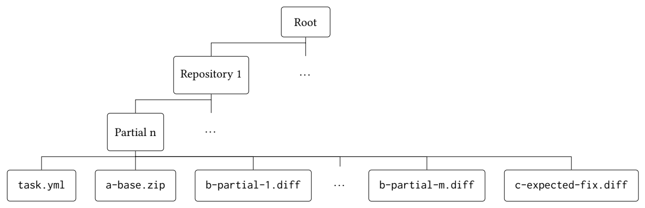

Figure 4: Directory and file structure of the created partial-fix dataset, taken from [5]

The sequences of commits can only be accepted as candidates, since the developers only indicate the bug-inducing commit. This may generate a sequence where there are multiple unrelated bugs being fixed.

2.2 Manual Candidate Validation

The candidate selection yields 2 362 candidates from GitHub issues and 5 999 candidates from the Linux kernel. We equally distribute all candidates from GitHub issues, of which we managed to look at 1 605, and 100 randomly selected issues from the Linux kernel to three authors for manual validation. All other candidates are discarded. Every author confirms or rejects a candidate as 'partial fix'. To increase the confidence in the classification, and to keep the workload manageable, 100 % of the confirmed but only 5 % of the rejected candidates of an author go to the other two authors for an additional inspection. A candidate is only labeled as 'partial fix' in the task-definition file (see Fig. 5) described in Sect. 2.3, if all authors agree on it being a 'partial fix'. If all authors reject a candidate, we label it as 'no partial fix' in the dataset. Candidates with less than three or mixed classifications are labeled as 'unknown'. The standard procedure for every candidate is as follows: (*i*) we first read the issue description and commit messages to get an idea of the issue, (*ii*) we then check all patches related to the issue and perform a qualitative analysis of the changes, and finally, (*iii*) we confirm or reject a candidate with a short explanation.

2.3 Organization of the Dataset

Format. Figure 4 depicts the directory and file structure of the dataset. The root directory contains a directory for each repository where at least one candidate has been labeled. We place a directory named 'Partial *n*' for every candidate in the directory of the respective repository. Initially, these only contain the task-definition file (cf. Fig. 5). However, it is possible to restore the base version before the first commit of the repository (a-base.zip), the diffs of all fix attempts (b-partial-*n*.diff), and the diff of the expected fix (c-expected-fix.diff) with the help of a provided script.

Task Definition. The task definition as seen in Fig. 5 lists all relevant information about a candidate in a machine- and human-readable format. After the format version, we give the URL of the repository followed by a sequence of commits, namely the base version, fix attempts, and the expected fix. Each commit consists of its commit hash and the input file. The base version points to a ZIP archive relative to the task definition containing the project at the given revision. The fix attempts and the expected fix point to a diff files. The diffs need to be applied sequentially to the base version to obtain the 'fixed' project. The classification has one of three values:

```
# Version of this task-definition format
format_version: '1.0'

# URL of the base repository with the bug
repository_url: github.com/util-linux/util-linux

# Information supposed to be given to the tool
sequence:
    # Buggy base version of the program
    base_version:
        # Archive of program
        input_file: 'c6b0cbdd.zip'
        # Commit hash of the base version
        commit-sha1: c6b0cbdd...

    # Fix attempts
    fix_attempt:
        - input_file: 'b6b5272b.diff'
          commit-sha1: b6b5272b...

    # Expected fix
    expected_fix:
        input_file: 'd8bfcb4c.diff'
        commit-sha1: d8bfcb4c...

# Does this task contain a partial fix?
classification: partial fix
# If yes, what kind of partial fix is it?
category: ["arithmetic␣and␣control-flow"]

# Metadata about the task
metadata:
    # Language of the project
    language: C
    # Which heuristic generated the candidate?
    strategy: reopen
    # Is there a build system available?
    build_system: ['make']
    # URL to the issue that describes the related bug
    related_issue: ..util-linux/util-linux/issues/422
```

Figure 5: The task-definition of our example in Sect. 1; slightly modified for brevity

partial fix, *no partial fix*, or *unknown* depending on the candidate being classified as a partial fix or not. If the authors disagree or did not yet inspect the candidate, we label it as *unknown*. Partial fixes also come with categories (cf. Table 1) and metadata, such as the programming language, the heuristic that generated the candidate, the build system, and optionally, the related issue.

3 THE P3 DATASET

Crawling GitHub issues for over 3 months, we collected 2 362 candidates in 3 717 crawled repositories, where 498 repositories contained at least one candidate. Figure 6 shows some statistics about the repositories containing at least one candidate. This demonstrates that the candidates come from a wide variety of projects in terms of stars, issues, and size. In total, we manually inspected 21 489 commits with over four million changes. We found 258 partial fixes in 2 362 issues of which we inspected 1 605. Our two criteria C_{reopen} and C_{status} matched 2 141 and 233 issues, respectively. In the Linux kernel we found 5 999 candidates of which we manually inspected 100 candidates and confirmed 40 partial fixes. When we tried to archive the repositories, `git` could not find some revisions of our candidates anymore. We excluded these candidates from the final dataset.

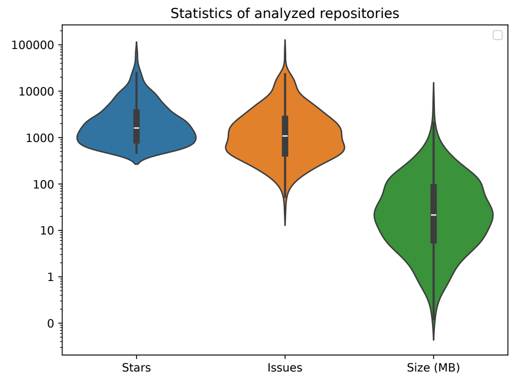

Figure 6: Distribution of received stars, number of issues, and size of considered repositories

Table 1: Classification of partial fixes

Keyword	#	Description
Arithmetic and Control-Flow	62	branching conditions, iteration amounts, off-by-one errors, . . .
Hardware and OS	14	problems with specific hardware or OS
Null Pointers	11	dereference, null check, . . .
Performance	10	time and memory consumption
Memory	8	memory leaks, segmentation faults, . . .
Multi-processing	5	race conditions, deadlocks, . . .
Wrong Data Type	5	overflow, can hold too few elements, . . .
Same Fix, Different Location	5	porting fix, copied code, . . .
Code Quality	5	refactoring, . . .
Preprocessing Directives	4	conditional compilation, declarations, . . .

In total, out of the original 1 705 candidates, we are left with 187 true 'partial fixes', 15 definite 'no partial fixes', and 1 116 unclassified or unknown candidates. The remaining 387 candidates were no longer available to download three months after crawling GitHub.

Partial-Fix Categories. To understand partial fixes better, we categorize them manually, assigning each a possibly empty set of descriptive labels from Table 1. Most of the partial fixes are related to arithmetic and control-flow operations. These are usually caused by missing or wrong conditions in if-statements or loops or off-by-one errors. The second most common category is related to hardware or operating-system-specific code. This includes code that is only executed on a specific operating system or hardware, and therefore cannot be tested on other systems. Other common categories are related to null pointers, performance, memory problems, multi-processing issues, and wrong data types. One particularly interesting category consists of partial fixes where the same fix needs to be applied in multiple locations, for example when back-porting code or when code was copied originally from another place.

Limitations. The dataset consists of patches in considerably large software projects. This may be challenging for some applications. The used projects are usually under active development. Therefore, between every commit in a task-definition file, unrelated parts of the code may have changed. Moreover, forcefully pushed commits or large merge commits from forks may introduce additional noise.

Future Work. We plan to continuously extend and refine the dataset by including projects in other programming languages, adding more partial fixes, and improving the automated selection process with a preliminary automated classification. Moreover, we want to isolate partial fixes by exposing the partial fix in a sequence of single files rather than a sequence of patches in a project.

4 THREATS TO VALIDITY

Internal validity. Since the crawler is based on heuristics, we miss some true partial fixes for certain projects. However, the quality of the benchmark set is not affected by this, since we manually inspect every candidate for partial fixes. Our manual cross-validation safeguards against the usual threats such as human error, bias, and performance. Therefore, we are confident of the correctness of this collection of partial fixes.

External validity. While our aim is to provide a diverse and representative set of partial fixes for the benchmark set, there is no guarantee that all possible classes of partial fixes have been covered. This is mitigated by crawling a wide variety of projects.

5 RELATED DATASETS

For bug finding, there are a many tools available and also several benchmark datasets [8, 14]. Further benchmarks have been proposed and used to evaluate automated fault-localization and program-repair techniques [6, 7, 9, 17, 19, 20, 23]. `DBGBench` [6] is a dataset with 27 real bugs aimed at evaluating new debugging approaches. The datasets `ManyBugs` and `IntroClass` [9] collectively contain 1 183 defects from 15 C programs. More widely used datasets include `Defects4J` [11], `BugsInPy` [22], `Space` [21], and the Siemens suite of small C programs. While some subject programs in these sets may contain partial fixes, their original intent does not readily lend themselves to addressing the incomplete-patch problem. Perhaps more related to ours is the recent dataset `ReCover` [1] of 28 Java programs designed to complement older datasets used for evaluating regression testing approaches. However, regression testing is concerned with a different category of bad fixes, while our focus here is exclusively on partial fixes.

6 CONCLUSION

This paper introduces and describes the dataset P3 for *partial* or *incomplete* patches. We collected partial-fix candidates from GitHub using heuristics and from the Linux-kernel repository. These candidates were then manually validated to ensure that they are indeed partial fixes. The raw data contain 258 labeled partial fixes and 32 false positives from which we derived 187 definitive partial fixes and 15 cases that are no partial fixes, in the final dataset.

Data-Availability Statement. The dataset is publicly available at https://gitlab.com/sosy-lab/research/data/partial-fix-dataset. The current snapshot is archived [3]. Additionally, a reproduction package including all scripts and our raw data is archived [4].

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY), 418257054 (Coop), and 496588242 (IdeFix).

REFERENCES

[1] F. Altiero, A. Corazza, S. Di Martino, A. Peron, and L. L. L. Starace. 2022. ReCover: A Curated Dataset for Regression-Testing Research. In *Proc. MSR*. ACM, 196–200. https://doi.org/10.1145/3524842.3528490
[2] L. An, F. Khomh, and B. Adams. 2014. Supplementary Bug Fixes vs. Re-opened Bugs. In *Proc. SCAM*. IEEE, 205–214. https://doi.org/10.1109/SCAM.2014.29
[3] D. Beyer, L. Grunske, M. Kettl, M. Lingsch-Rosenfeld, and M. Raselimo. 2023. Partial-Fix Benchmarks: A Benchmark Set for Program Repair on Partial Fixes. Zenodo. https://doi.org/10.5281/zenodo.10369427
[4] D. Beyer, L. Grunske, M. Kettl, M. Lingsch-Rosenfeld, and M. Raselimo. 2023. Reproduction Package for MSR 2024 Article 'P3: A Dataset of Partial Program Fixes'. Zenodo. https://doi.org/10.5281/zenodo.10319627
[5] D. Beyer, L. Grunske, T. Lemberger, and M. Tang. 2021. Towards a Benchmark Set for Program Repair Based on Partial Fixes. *arXiv/CoRR* 2107, 08038 (July 2021). http://arxiv.org/abs/2107.08038.
[6] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller. 2017. Where Is the Bug and How Is It Fixed? An Experiment with Practitioners. In *Proc. FSE*. ACM, 117–128. https://doi.org/10.1145/3106237.3106255
[7] D. Callaghan and B. Fischer. 2023. Improving Spectrum-Based Localization of Multiple Faults by Iterative Test-Suite Reduction. In *Proc. ISSTA*. ACM, 1445–1457. https://doi.org/10.1145/3597926.3598148
[8] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz. 2009. BegBunch: Benchmarking for C Bug-Detection Tools. In *Proc. DEFECTS*. ACM, 16–20. https://doi.org/10.1145/1555860.1555866
[9] C. Le Goues, N. J. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Trans. Softw. Eng.* 41, 12 (2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513
[10] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. 2010. Has the Bug Really Been Fixed?. In *Proc. ICSE*. ACM, 55–64. https://doi.org/10.1145/1806799.1806812
[11] R. Just, D. Jalali, and M. D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proc. ISSTA*. ACM, 437–440. https://doi.org/10.1145/2610384.2628055
[12] S. Kim, K. Pan, and E. J. Whitehead Jr. 2006. Memories of Bug Fixes. In *Proc. FSE*. ACM, 35–45. https://doi.org/10.1145/1181775.1181781
[13] H. K. N. Leung and L. J. White. 1989. Insights into Regression Testing (Software Testing). In *Proc. ICSM*. IEEE, 60–69. https://doi.org/10.1109/ICSM.1989.65194
[14] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. 2005. Bugbench: Benchmarks for Evaluating Bug-Detection Tools. In *Proc. Workshop on the Evaluation of Software Defect Detection Tools*, Vol. 5.
[15] A. Mockus and D. M. Weiss. 2000. Predicting Risk of Software Changes. *Bell Labs Tech. J.* 5, 2 (2000), 169–180. https://doi.org/10.1002/BLTJ.2229
[16] A. K. Onoma, W.-T. Tsai, M. H. Poonawala, and H. Suganuma. 1998. Regression Testing in an Industrial Environment. *Commun. ACM* 41, 5 (1998), 81–86. https://doi.org/10.1145/274946.274960
[17] P. Orvalho, M. Janota, and V. Manquinho. 2022. C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments. *arXiv/CoRR* 2206, 08768 (June 2022). https://doi.org/10.48550/arXiv.2206.08768
[18] J. Park, M. Kim, B. Ray, and D.-H. Bae. 2012. An Empirical Study of Supplementary Bug Fixes. In *Proc. MSR*. IEEE, 40–49. https://doi.org/10.1109/MSR.2012.6224298
[19] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury. 2018. Repairing Crashes in Android Apps. In *Proc. ICSE*. 187–198. https://doi.org/10.1145/3180155.3180243
[20] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury. 2017. Codeflaws: A Programming-Competition Benchmark for Evaluating Automated Program-Repair Tools. In *Proc. ICSE Companion*. IEEE, 180–182. https://doi.org/10.1109/ICSE-C.2017.76
[21] F. I. Vokolos and P. G. Frankl. 1998. Empirical Evaluation of the Textual Differencing Regression-Testing Technique. In *Proc. ICSM*. IEEE, 44–53. https://doi.org/10.1109/ICSM.1998.738488
[22] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh. 2020. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *Proc. ESEC/FSE*. ACM, 1556–1560. https://doi.org/10.1145/3368089.3417943
[23] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury. 2017. A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments. In *Proc. FSE*. ACM, 740–751. https://doi.org/10.1145/3106237.3106262
[24] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram. 2011. How Do Fixes Become Bugs?. In *Proc. ESEC/FSE*. ACM, 26–36. https://doi.org/10.1145/2025113.2025121
[25] A. Zeller and R. Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498