TrickyBugs: A Dataset of Corner-case Bugs in Plausible Programs

Kaibo Liu
liukb@pku.edu.cn
Peking University
Beijing, China

Yudong Han
hanyd@pku.edu.cn
Peking University
Beijing, China

Yiyang Liu
ptr1479@stu.pku.edu.cn
Peking University
Beijing, China

Jie M. Zhang
jie.zhang@kcl.ac.uk
King's College London
London, United Kingdom

Zhenpeng Chen
zp.chen@ucl.ac.uk
University College London
London, United Kingdom

Federica Sarro
f.sarro@ucl.ac.uk
University College London
London, United Kingdom

Gang Huang
hg@pku.edu.cn
Peking University
National Key Laboratory of Data Space Technology and System
Beijing, China

Yun Ma
mayun@pku.edu.cn
Peking University
Beijing, China

ABSTRACT

We call a program that passes existing tests but still contains bugs as a buggy plausible program. Bugs in such a program can bypass the testing environment and enter the production environment, causing unpredictable consequences. Therefore, discovering and fixing such bugs is a fundamental and critical problem. However, no existing bug dataset is purposed to collect this kind of bug, posing significant obstacles to relevant research. To address this gap, we introduce TrickyBugs, a bug dataset with 3,043 buggy plausible programs sourced from human-written submissions of 324 real-world competition coding tasks. We identified the buggy plausible programs from approximately 400,000 submissions, and all the bugs in TrickyBugs were not previously detected. We hope that TrickyBugs can effectively facilitate research in the fields of automated program repair, fault localization, test generation, and test adequacy.

CCS CONCEPTS

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

KEYWORDS

software testing, test generation, test adequacy, program repair, benchmark

ACM Reference Format:
Kaibo Liu, Yudong Han, Yiyang Liu, Jie M. Zhang, Zhenpeng Chen, Federica Sarro, Gang Huang, and Yun Ma. 2024. TrickyBugs: A Dataset of Corner-case Bugs in Plausible Programs. In *Proceedings of 21st International Conference on Mining Software Repositories (MSR 2024).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

"Testing shows the presence, not the absence of bugs."
—Edsger W. Dijkstra

Testing is a critical step in ensuring software quality. However, tests are no proof of correctness. This is because tests represent an incomplete specification of the program requirements, and limited test cases may miss certain corner cases.

We refer to a program that passes existing tests as a *plausible program*, which may either be correct or buggy. Bugs in plausible programs are typically harder to detect because plausible programs exhibit correct behavior on existing test cases and only manifest incorrect behavior on specific situations, which are often logical corner cases. These undetected corner-case bugs can potentially pass all tests in the testing environment and make their way into the production environment, which may lead to unforeseen losses.

Existing bug datasets like Defects4J [3], CodeFlaws [14], and QuxixBugs [7] have been widely used in the fields of automated program repair and fault localization. These techniques typically take a buggy program along with the test cases it passes and fails as input, where the failed test cases are often crucial. Therefore, most of the bugs in these datasets were identified by a simple and ordinary test case and do not necessarily represent corner cases.

To address this gap, we present TrickyBugs, a dataset of 3,043 corner-case bugs in plausible programs sourced from 324 real-world competition coding tasks. The bugs in Trickybugs were all previously undiscovered, and finding or fixing these bugs can be more challenging.

TrickyBugs proposes a more practical and challenging problem: how to fix and locate faults **in the absence of failed test cases**, with only source code, passed test cases, and program specification available? There have been several discussions regarding similar problems in recent research [5, 15]. Trickybugs provides an appropriate and effective evaluation dataset for such tasks.

Trickybugs can also facilitate research related to **test adequacy** and **unit test generation**. The inability of the original test cases to discover bugs in TrickyBugs is, in fact, an issue related to test adequacy. As mentioned in our prior work [8], traditional test adequacy metrics such as line/branch coverage and mutation score have limitations when applied to TrickyBugs. Consequently, TrickyBugs can be utilized to study how to better measure and improve test adequacy. Similarly, TrickyBugs is suitable for evaluating tasks of generating failure-inducing test cases based on source code and program specification.

TrickyBugs is publicly available at https://github.com/RinCloud/TrickyBugs/.

2 DATASET CONSTRUCTION

In this section, we first describe how we collect the programs and find the buggy plausible programs by test generation, and then we describe how we equip some of the buggy plausible programs with their fixed versions.

2.1 Data Source

We collect our data from AtCoder, a highly popular and active programming competition platform that has been used as a reliable resource for different research purposes in the domain of computer science [6, 12, 16]. In AtCoder, users are requested to submit their code solutions for different coding programs, and the backend test cases will judge the submitted programs. A test case is an input/output pair, and if a program's output matches the test output exactly when fed with the corresponding test input, it passes the test case; otherwise, it fails.

We collect AtCoder submissions from CodeContests [6], where the duplicate programs were removed. These submissions primarily come from the years 2016 to 2021. As mentioned in Section 1, we are not interested in bugs that the existing tests can detect; instead, we are interested in the bugs that have not been discovered. Therefore, we only retain the passing programs. At this step, we get approximately 230,000 human-written programs in C++, 140,000 programs in Java, and 169,000 programs in Python from 939 coding tasks, and then we will find the hidden bugs among these programs. We also collect the original test cases and difficulty of the coding tasks from an AtCoder's official post [2] and a third-party website that evaluates the difficulty of coding tasks based on AtCoder's rating system [11], respectively.

2.2 Bug Detection

After collecting a large number of plausible programs, the next step is to find bugs among them. This is the core step for dataset construction.

In summary, we randomly generate valid test inputs for each coding program. Then, we address the test oracle problem through differential testing to obtain test outputs. These test outputs and their corresponding test inputs together form additional test cases. At last, these additional test cases are used to identify buggy plausible programs.

2.2.1 Test Input Generation. Each coding task in AtCoder provides detailed constraints for inputs in its problem description. Any test input fulfilling the constraints is a valid input. To generate such valid test inputs, we manually write random test generator scripts using the Python library crayon [9] for each coding task according to its input constraints. The generators can produce valid test inputs in a relatively uniform random manner. For example, to generate a random integer, the generator samples the integer uniformly from the range specified in the input constraints. We generate one hundred additional test inputs for each coding task.

2.2.2 Test Oracle Generation. The test oracle problem refers to the challenge of determining whether a program's behavior is correct. For example, given a test input, it involves figuring out whether the program's output is correct. This is indeed a classic and challenging problem in the field of software engineering. To tackle the test oracle problem, we employ the technique of differential testing [1], a software testing technique that involves comparing the output of two or more different implementations of the same functionality to identify discrepancies between them.

To ensure the validity of differential testing, we need to filter out coding tasks with multiple correct outputs for the same input, retaining coding tasks with only one correct test output for the same test input. Therefore, for any valid test input, discrepancies among outputs indicate the presence of at least one incorrect output, and the programs producing the incorrect output are bound to be buggy. In this step, we filtered out 48 (5.1%) unsuitable coding tasks and proceeded with the remaining 891 coding tasks for the subsequent steps.

Then, it comes to the process of differential testing. We feed the same generated test input to all the plausible programs of the same coding task and collect their outputs. If there is any discrepancy among the outputs, we find bugs. We designate the output that dominates the others in terms of proportion as the correct test output. We also call this output the major output, and any output that differs from the major output is incorrect, and the programs that produce the incorrect outputs are buggy plausible programs. So far, we have successfully identified bugs in plausible programs.

2.3 Bug Repair

We also provide the fixed version for some of the buggy plausible programs. The fixed programs can be used for diverse software engineering tasks such as fault localization. Due to the large dataset scale and the various difficulties of the coding tasks, we utilize Automated Program Repair (APR) techniques to assist us in fixing these buggy plausible programs. According to recent studies [13, 17], APR techniques based on Large Language Models (LLM) have demonstrated state-of-the-art performance. Therefore, we employ the `gpt-3.5-turbo` model for the APR task. We combine the buggy plausible program with the corresponding coding task's problem description as part of the prompt provided to LLM, requesting the LLM to generate a fixed version of the program. If a fixed program can pass both the original test cases and additional test cases, we will proceed to verify whether this fixed program is correct manually. If the fixed program passes the manual verification, it is considered valid and will serve as the fixed version of the corresponding buggy program in the dataset.

3 TRICKYBUGS DATASET

In this section, we introduce TrickyBugs dataset, a dataset of buggy plausible programs. This dataset contains 1,405 buggy programs in C++, 792 in Java, and 846 in Python from 324 coding tasks. We also provide 1,361 fixed programs from 224 coding tasks to broaden the applicability of the dataset.

3.1 Data Structure

TrickyBugs dataset contains root directories of each coding task, and each root directory is named as the `pid` of the coding task, which uniquely identifies the coding task within the dataset.

The root directory of each coding task contains the following files and subdirectories:

(1) `buggy_programs`: A directory. This directory contains all the buggy plausible programs we found. Programs in different languages (C++, Java, or Python) are stored in separate directories.
(2) `reference_programs`: A directory. This directory contains reference programs in C++. A reference program always produces the major output for any test input throughout the process of our differential testing. Reference programs are considered correct. We provide multiple reference programs (up to five) because they can be used for preliminary verification of the validity of an input. Specifically, for any given test input, if all reference programs produce the same output, then this test input is likely valid. Otherwise, it is highly probable to be invalid.
(3) `fixed_programs` (optional): A directory. This directory contains the fixed version of some buggy plausible programs. These fixed programs are useful for fault localization. It's important to note that many bugs in TrickyBugs originate from logical corner cases. Therefore, the differences between the buggy program and its fixed version may not be limited to just one line but could involve multiple lines. Not every buggy program has a fixed version, and 224 out of 324 root directories of coding tasks contain this subdirectory.
(4) `original_test_cases` (optional): A directory. This directory contains all the original test cases for the coding task on AtCoder, and 274 out of 324 root directories of coding tasks contain this subdirectory because AtCoder has not publicly disclosed the test cases for some earlier coding tasks [2]. However, it is still possible and easy to determine whether a program passes the original test cases by submitting the code on AtCoder. The submission URL corresponding to the coding task is provided in the `metainfo.json`. A plausible program, whether they are buggy or not, should pass all the original test cases.
(5) `additional_test_cases`: A directory. This directory contains the additional test cases that have uncovered bugs successfully. A bug-free program should pass all the original and additional test cases.
(6) `metainfo.json`: A file. This file contains serval meta information of the coding task. The *URL* is the source of this coding task, which contains all the information about this coding task except the original test cases. The `test program mapping` displays the mapping between an additional test case and the buggy plausible programs it has identified.
(7) `problem_description.txt`: A file. This file contains the problem description, input constraints, and several pairs of input/output examples for this coding task. It is a file version of the content from the URL of the coding task and represents a detailed program specification for the coding task.

Table 1: Statistics of TrickyBugs.

Feature	C++	Java	Python	Total
# Programs	1,405	792	846	3051
# Tasks	251	159	115	324
Average LOC	39.4	74.7	13.9	41.5
Average Diff.	1469.1	865.4	907.5	1098.9
Average # original tests	/	/	/	28.2
Average # additional tests	/	/	/	2.3

3.2 Statistics

In this section, we introduce more details about TrickyBugs. Table 1 presents some basic statistical information about TrickyBugs. This table shows the number of buggy plausible programs (# Programs), the number of coding tasks that contain the buggy plausible programs (# Tasks), the average lines of code (Average LOC), the average difficulty of the coding tasks (Average Diff.), the average number of original test cases (Average # original tests) and additional test cases (Average # additional tests) per coding task. The average number and difficulty of C++ buggy plausible programs are both the highest. This may be because C++ offers faster execution speed and is the most mainstream language in program competitions. Additionally, we also observed that each coding task contains 28.2 original test cases on average. Despite this relatively large size of tests (for one single functionality), many bugs remain undiscovered, which indicates the challenging nature of discovering these bugs.

4 DISCUSSION

4.1 Validity of the Additional Test Cases

For any coding task, as long as a test input is valid (it satisfies all the constraints specified by the coding task), the test case formed by the pair of this input and its corresponding output is valid. The validity of the test generators is confirmed manually, and the correctness of the test outputs is based on the majority rule. For example, if 99 plausible programs output “yes" and only 1 plausible program outputs “no" for the same input, it is highly likely that “yes" is the correct output. In fact, most of the situations we encounter during the step of test oracle construction are quite similar to this example. For a given test input, we refer to the proportion occupied by the output with the highest occurrence among all obtained outputs as the **dominance ratio**. A larger dominance ratio often indicates a more reliable test output. For the example we just described, the dominance ratio is 0.99.

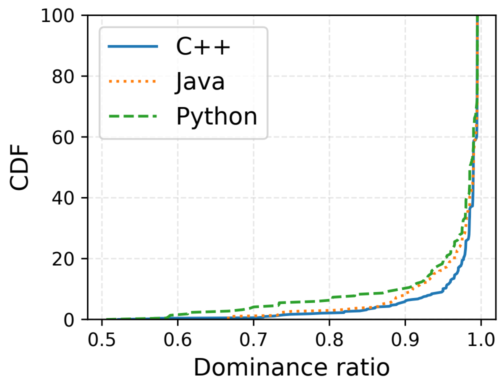

Figure 1: CDF of the dominance ratio of the additional test inputs in TrickyBugs. For the three languages C++, Java, and Python, the respective proportions of additional test inputs with a dominance ratio greater than 0.95 are 89.7%, 83.1%, and 80.0%.

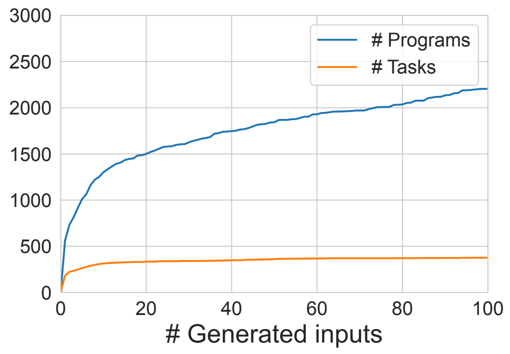

Figure 2: Trendency of the number of discovered buggy plausible programs (# Programs) and the number of coding tasks that contain the programs (# Tasks) in tandem with the increase in the number of generated test inputs.

Figure 1 shows the cumulative distribution function (CDF) of the dominance ratio of all additional test inputs in TrickyBugs. For the three languages C++, Java, and Python, the respective proportions of additional test inputs with a dominance ratio greater than than 0.95 are 89.7%, 83.1%, and 80.0%, which indicates that the corresponding test outputs are highly reliable. We also manually check all the test cases whose dominance ratio are lower than 0.95 to guarantee their validity.

4.2 Adequacy of the Additional Test Cases

In Section 2.2.1, we generate 100 test inputs for each coding task, and a possible question is whether 100 generated test inputs are adequate to identify buggy plausible programs. To explore this question, we illustrate how the number of generated inputs influences the number of discovered buggy plausible programs and the number of coding tasks that contain the programs in Figure 2. The x-axis is the number of generated inputs. The blue line is the number of discovered buggy plausible programs, which reaches a plateau when there are around 95 inputs. The orange line is the number of coding tasks that are detected to contain buggy plausible programs, which reaches a plateau when there are around 15 inputs. These observations indicate that generating 100 test inputs is generally adequate to reach the upper limit of the capability of bug discovery within our methodology.

4.3 Originality of TrickyBugs

We proposed TrickyBugs as a part of an empirical study of the test cases of online coding tasks [8]. Since then, we have conducted a comprehensive upgrade of the TrickyBugs dataset. We added reference programs for each coding task and added 1,361 fixed programs. We also filtered out some programs that did not quite align with the definition of plausible buggy programs due to issues related to uninitialized variables in C++. In summary, this new version of TrickyBugs is more comprehensive and sound, making it a valuable resource for research in various fields of software engineering.

5 RELATED WORK

There are numerous datasets of software bugs available to support research in the field of software engineering. **Defects4J**[3] is a database providing real bugs from open-source programs in Java to enable reproducible studies in software testing research. **ManyBugs** and **IntroClass** [4] collect defects of C programs from student programming assignments and open-source projects. **BugSwarm** is a collection of thousands of real software bugs and corresponding fixes. **Codeflaws**[14] collects buggy programs from Codeforces platform [10] and categorized these bugs based on program syntax. **QuixBugs** [7] is a dataset of buggy programs in both Java and Python based on 40 coding tasks from the Quixey Challenge. **ConDefects** [16] is a dataset with buggy programs extracted from recent AtCoder coding tasks to address the issue of data leakage when evaluating the coding capabilities of large language models.

Most of the bugs in the existing datasets have been discovered by the existing test cases before they were collected. In contrast, the bugs in Trickybugs were all previously undiscovered, which suggests that the bugs are more challenging to detect and represent more corner cases.

6 CONCLUSION

We introduce TrickyBugs, a dataset of 3,043 buggy plausible programs sourced from real-world submissions of 324 coding tasks. The bugs in TrickyBugs were all previously undiscovered. TrickyBugs is suitable for research related to program repair, fault localization, test adequacy, and test generation. We will continue to update and expand TrickyBugs, and we hope that TrickyBugs can effectively contribute to the software engineering community.

REFERENCES

[1] Robert B Evans and Alberto Savoia. 2007. Differential testing: a new approach to change detection. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*. 549–552. https://doi.org/10.1145/1295014.1295038

[2] AtCoder Inc. 2016. Atcoder testcases. https://atcoder.jp/posts/21

[3] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

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

[5] Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 14–26.

[6] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158 arXiv:https://www.science.org/doi/pdf/10.1126/science.abq1158

[7] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*. 55–56.

[8] Kaibo Liu, Yudong Han, Jie M Zhang, Zhenpeng Chen, Federica Sarro, Mark Harman, Gang Huang, and Yun Ma. 2023. Who Judges the Judge: An Empirical Study on Online Judge Tests. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[9] luogu dev. 2019. Cyaron. https://github.com/luogu-dev/cyaron.

[10] Mikhail Mirzayanov. 2009. Codeforces. https://codeforces.com/

[11] Kenko Nakamura. 2015. Atcoder problems models. https://kenkoooo.com/atcoder/resources/problem-models.json

[12] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655* 1035 (2021).

[13] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653* (2023).

[14] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. 2017. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 180–182.

[15] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2023. Software testing with large language model: Survey, landscape, and vision. *arXiv preprint arXiv:2307.07221* (2023).

[16] Yonghao Wu, Zheng Li, Jie M Zhang, and Yong Liu. 2023. ConDefects: A New Dataset to Address the Data Leakage Concern for LLM-based Fault Localization and Program Repair. *arXiv preprint arXiv:2310.16253* (2023).

[17] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*.