BugsC++: A Highly Usable Real World Defect Benchmark for C/C++

Gabin An	Minhyuk Kwon	Kyunghwa Choi	Jooyong Yi	Shin Yoo
KAIST	*Suresoft Technologies*	*Suresoft Technologies*	*UNIST*	*KAIST*
Republic of Korea	Republic of Korea	Republic of Korea	Republic of Korea	Republic of Korea
agb94@kaist.ac.kr	minhyuk@suresofttech.com	khchoi@suresofttech.com	jooyong@unist.ac.kr	shin.yoo@kaist.ac.kr

***Abstract*—As software systems grow larger and more complex, debugging takes up an increasingly significant portion of developers' time and efforts during software maintenance. To aid software engineers in debugging, many automated debugging and repair techniques have been proposed. Both the development and evaluation of these automated techniques depend on benchmarks of bugs. While many different defect benchmarks have been developed, only a few benchmarks are widely used due to the origin of the collected bugs as well as the usability of the benchmarks themselves, risking a biased research landscape. This paper presents BugsC++, a new benchmark that contains 209 real-world bugs collected from 22 open-source C/C++ projects. BugsC++ aims to provide high usability by providing a similar user interface to the widely used Defects4J. Further, BugsC++ ensures the replicability of the bugs in its collection by encapsulating each buggy program in a Docker container. By providing a highly usable real-world defect benchmark for C/C++, we hope to promote debugging research for C/C++.**

***Index Terms*—software testing, bug, fault, defect benchmark**

I. Introduction

Software developers spend a lot of time testing and fixing existing code [1]. As the size and complexity of a program inevitably increase the cost of its maintenance, debugging techniques such as Fault Localization (FL) [2] or Automated Program Repair (APR) [3] have been actively researched to increase developer productivity and reduce the debugging cost.

Various defect data have been used to quantitatively evaluate and compare the effectiveness and efficiency of automatic debugging techniques, among which Defects4J [4], a collection of reproducible real-world defects in open-source Java programs, and SIR [5] or Siemens [6], a collection of defects in C programs, are the most widely used. Recent evaluations of automatic debugging methodologies have been mainly focused on specific benchmarks, which has the advantage of making it easier to compare different methodologies, but there is also a concern that if the evaluation of techniques is focused on a few defect benchmarks for a long time, the development of the technology may be over-optimized for those benchmarks. To avoid this problem, it has been suggested in the literature that different kinds of benchmarks should be used when evaluating automatic debugging techniques [3], and several software engineering journals have encouraged replicability studies, where existing techniques are evaluated on new data.

In this paper, we propose BugsC++ (pronounced as *bugsy*, with '++' omitted for simplicity), a benchmark that consists of reproducible defects collected from various open-source C/C++ programs, to increase the diversity of the defect benchmark ecosystem. Based on the observation that most of the popular benchmarks have high usability, we simplify the usage of essential features such as test execution/coverage measurement to increase the usability of BugsC++. We also use the virtualization application Docker [9] to facilitate the environment setup for benchmark execution as well as to ensure the reproducibility of defects.

TABLE I: Benchmarks Listed in https://program-repair.org

Name	Language/ Platform	Real Bugs? (# bugs, # projects)
DroixBench [7]	Android	Yes (24, 15)
C-Pack-IPAs [8]	C	No†
CodeFlaws [9]	C	No‡
DBGBench [10]	C	Yes (27, 2)
ITSP [11]	C	No†
IntroClass [12]	C	No†
ManyBugs [12]	C	Yes (185, 9)
Bears [13]	Java	Yes (251, 72)
Bugs.jar [14]	Java	Yes (1,158, 8)
Vul4J [15]	Java	Yes (79, 52)
Defects4J v2.0 [4]	Java	Yes (835, 17)
BugsJS [16]	JavaScript	Yes (453, 10)
FixJS [17]	JavaScript	Yes (323,907, -)
BugsInPy [18]	Python	Yes (493, 17)
Refactory [19]	Python	No†
Defexts [20]	Kotlin, Groovy, Scala	Yes (654, 413)
BugSwarm [21]	Python, Java	Yes (2,611, -)
QuixBugs [22]	Python, Java	No‡

†: Class Assignments, ‡: Programming Competition Submissions

II. Existing Bug Benchmarks

Table I lists defect benchmarks reported by https://program-repair.org, a repository that curates bibliography, tools, and benchmarks related to APR research (as of 23 May 2023). We will restrict our discussion of existing defect benchmarks to the ones listed in Table I; widely used, yet older benchmarks such as SIR [5] and Siemens Suite [6] contain programs that are too small, and bugs that are artificially injected, hence are excluded from our discussion.

As of May 2023, there are six C benchmarks, compared to four Java benchmarks (excluding the multilingual ones that include Java, e.g., BugSwarm [21] or QuixBugs [22]), other than BugsC++. However, only two of these C benchmarks contain bugs collected from real-world projects (DBGBench [10], which is an extension of COREBench [23], and ManyBugs [12]), whereas all four Java benchmarks consist of real-world bugs. Further, these two C benchmarks consist only of 11 projects in total, whereas the average number of projects in Java benchmarks is significantly higher, 37.25. The comparison suggests that we still need C benchmarks that contain bugs from more diverse projects.

We have also analyzed the defect benchmarks used by APR-related papers published in 2022 based on the list at https://program-repair.org/bibliography.html: there are currently 64 publications. For each of these, we investigated which defect benchmark is used for the study; if multiple benchmarks are used, we counted all of them separately. In total, Java defect benchmarks appear in 28 publications, whereas C defect benchmarks appear in only eight publications. More importantly, out of the 28 publications that study Java defect benchmark, 18 contain Defects4J, showing its widespread adoption. While this is not a complete survey of APR publications from the year 2022, the sample from https://program-repair.org does show the dominance of Defects4J as the de facto standard benchmark, as well as the popularity of Java as the target language in APR research.

We cautiously attribute the popularity of Defects4J to a number of factors. Released in 2014, it had sufficient time to establish itself as the de-facto standard benchmark: more recent studies use Defects4J to compare their results to earlier work. However, age is definitely not the only reason. Defects4J is actively maintained, with each version adding more bugs to the benchmark. It also provides a convenient and powerful Command Line Interface (CLI), which facilitates easy experimentation. It is also well integrated with relevant tools such as the coverage profiler, `Cobertura`. Taking these factors into consideration, we set up the following design goals for our new benchmark:

- **Representativity**: The benchmark should consist of defects that are representative of real-world software [24].
- **Diversity**: The benchmark should encompass diverse defects from different projects, providing a realistic representation of real-world scenarios and allowing evaluation under various conditions.
- **Reproducibility**: The benchmark should enable the consistent reproduction of the buggy behavior of a program. This allows researchers and practitioners to evaluate and compare analysis and repair techniques reliably.
- **Usability**: The benchmark should be designed with ease of use in mind, providing convenient features such as a CLI for common use cases like test execution and coverage profiling. Usability ensures that the benchmark can be easily utilized by researchers and developers.

TABLE II: Projects and bugs included in BugsC++

		Average SLoC Across All Bugs		
Project	#. Bugs	C	C++	C/C++ Header
`coreutils`	2	58,178	0	2,948
`cpp_peglib`	10	0	2,037	12,671
`cppcheck`	30	0	60,289	7,328
`dlt_daemon`	1	33,303	1,360	1,945
`exiv2`	20	505	49,294	3,673
`jerryscript`	11	83,244	0	12,770
`libchewing`	8	7,484	0	185
`libssh`	1	42,356	0	39
`libtiff`	5	56,249	261	2,361
`libtiff_sanitizer`	4	56,597	261	2,365
`libucl`	6	6,421	0	706
`libxml2`	7	202,130	0	26,928
`md4c`	10	5,224	0	139
`ndpi`	4	36,056	7,282	13,510
`openssl`	28	278,756	0	44,341
`proj`	28	5,222	62,951	4,528
`wget2`	3	23,186	0	825
`wireshark`	6	3,246,700	0	140,261
`xbps`	5	19,629	0	1,017
`yaml_cpp`	10	0	4,345	4,380
`yara`	5	32,587	0	4,764
`zsh`	5	102,906	0	4,940
Total	209	4,296,733	188,080	292,632

The existing benchmarks for C/C++ programs are either less realistic because they only contain artificial bugs [6], [5], or less usable because they do not provide CLI for tasks such as test execution or coverage profiling [12].

III. BugsC++

We propose a new defect benchmark, BugsC++, that aligns with the design goals discussed earlier and serves as a comprehensive benchmark for evaluating automatic debugging techniques against C/C++ programs.

A. Project Selection and Data Collection

To ensure the **representativity** and **diversity**, our benchmark includes actual defects that were collected from code changes made to various open-source C/C++ projects, including standalone tools (e.g., `coreutils`, `cppcheck`, `wget2`), libraries (e.g., `libssh`, `libtiff`, `libxml`), parsers/runtimes (e.g., `jerryscript`, `md4c`, `yaml_cpp`), and user shell (`zsh`): refer to https://github.com/Suresoft-GLaDOS/bugscpp for more details.

We employ two criteria for choosing projects. First, we identify projects that have a revision history containing both a commit that fixed a bug and the corresponding test case. Second, we identify projects that have already reported Common Vulnerabilities and Exposures (CVEs)[1] and their associated fixes. This ensures that the defects included in BugsC++ have been recognized as security vulnerabilities or other significant issues in the software community. For the selection of defects to include, we use the following criteria: the buggy version of the project (1) should be successfully built on Ubuntu 18.04 and 20.04 (which we use for our Docker containers as described in Section III-B), (2) should have available test cases that can be executed, (3) should exhibit test failures

[1] https://cve.mitre.org/cve/search_cve_list.html

that disappear once patched, and (4) test coverage should be measurable using tools like gcov. By following this rigorous selection process, BUGSC++ comprises 209 real-world bugs collected from a total of 22 projects, as shown in Table II.

B. Dockerization

In comparison to Java, which is designed to be platform-independent, C/C++ programs are more susceptible to variations in behavior due to their reliance on different environments, such as compilers, operating systems, system libraries, and hardware. This can pose challenges to the reproducibility of defect benchmarks. To mitigate these issues and ensure consistent execution environments, we employ Docker, a lightweight virtualization technique, to build and run all target projects within a Docker container, thereby enhancing the **reproducibility** of the benchmark. In addition to improving reproducibility, Docker also enhances the **usability** of the benchmark. Users no longer need to go through the process of setting up separate environments for running the benchmark, reducing the associated setup costs and complexities. The utilization of Docker provides a standardized execution environment, simplifying the overall usability of the benchmark for researchers and practitioners.

C. CLI Commands & Examples

BUGSC++ provides CLIs that facilitate the building and execution of the included projects. By offering a user-friendly CLI, BUGSC++ ensures that researchers and practitioners can easily navigate and interact with the benchmark, i.e., the good **usability**. We note that the CLI commands are inspired by the features offered by Defects4J.

1) Checkout: To check out the code from project `P` that contains the `V`-th bug into the directory `D` (default: `./<P>`), one can use the `checkout` command.

```
> python bugscpp/bugscpp.py checkout <P> <V> --buggy
--target <D>
```

ex) Check out the first bug in the `zsh` project

```
> python bugscpp/bugscpp.py checkout zsh 1 --buggy
```

After the command is successfully executed, the source code can be found in the directory `<D>/buggy-<V>`. Furthermore, by omitting the option `--buggy`, one can get the fixed version of the bug, which will be saved to the directory `<D>/fixed-<V>`.

2) Build: The `build` command allows us to build the source code in the directory `<C>` (e.g., `<D>/buggy-<V>` or `<D>/fixed-<V>`) obtained by the checkout command:

```
> python bugscpp/bugscpp.py build <C>
```

ex) Build the first buggy version of the `zsh` project

```
python bugscpp/bugscpp.py build ./zsh/buggy-1/
```

3) Test: To execute the whole test suite after building a project, one can use the `test` command. To further ensure usability and also accommodate the requirements of test-based automated debugging techniques, BUGSC++ incorporates the capability to execute individual test cases separately for all projects, which allows for the independent extraction of results and dynamic information for each test case. In cases where projects do not support the execution of individual test cases (e.g., when all tests are called within a single function), we have made slight modifications to the test code, by inserting a switch statement in the test program, to enable the selection of individual tests by providing the test index as input.

```
> python bugscpp/bugscpp.py test <C> --output-dir <O>
```

ex) Run only a subset of test cases

```
> python bugscpp/bugscpp.py test <C> --output-dir <O>
--case <T1>,...,<Tn>
```

ex) Run the test cases `1` and `3` of `zsh-1`

```
> python bugscpp/bugscpp.py test ./zsh/buggy-1/
--output-dir ./zsh-results --case 1,3
```

The passed/failed results and the raw output of each test case `T` will be saved to `<O>/<P>-buggy-<V>-<T>/<T>.test` and `<O>/<P>-buggy-<V>-<T>/<T>.output`, respectively.

4) Coverage: When building code and running test cases, including the `--coverage` option will trigger the usage of the `gcov` coverage profiler to measure the coverage of each test case. Additionally, there is the flexibility to specify `gcov` options to obtain the output in the format a user prefers.

```
> python bugscpp/bugscpp.py build <C> --coverage
> python bugscpp/bugscpp.py test <C> --coverage
--output-dir <O>
```

ex) Specify the `gcov` options

```
> python bugscpp/bugscpp.py test <C> --coverage
--output-dir <O> --additional-gcov-options <OPTIONS>
```

The coverage results will be saved to the directory `<O>/<P>-buggy-<V>-<T>/gcov/`.

5) Search: To aid systematic evaluation in any experiment based on BUGSC++, we have added tags to each bug, based on the characteristics of each defect as well as its fix. Table III presents the categorization of tags along with the corresponding list of tags within each category. Furthermore, Figure 1

TABLE III: The available tags in BUGSC++

Tag Category	Tags
Buggy Lines	single-line, multi-line
Patch Types	addded, removed, modified
Error Types	invalid-condition, invalid-format-string, memory-error, logical-error, omission, infinite-loop-error, division-by-zero
Misc.	CVE, address-sanitizer

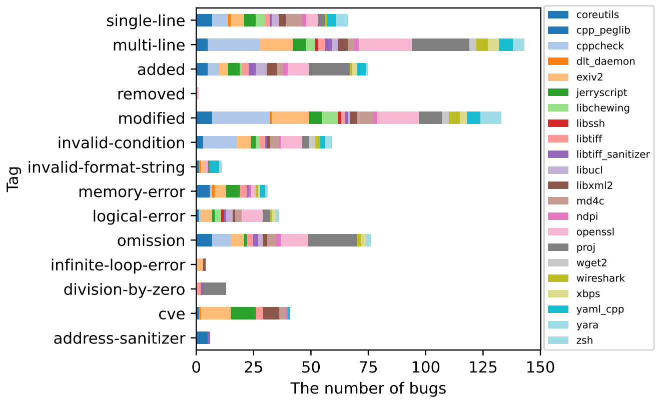

Fig. 1: Distribution of Bugs by Tag

provides an overview of the statistical summary for each tag in the dataset. To facilitate user search for specific types of defects, the `search` command can be used to extract only those defects that contain a specific tag `T`.

```
> python bugscpp/bugscpp.py search <T>
-------------------------------------------
ex) Find memory-related bugs
> python bugscpp/bugscpp.py search memory_error
```

More detailed user manuals, as well as descriptions for individual bugs, are publicly available at: https://github.com/Suresoft-GLaDOS/bugscpp.

IV. Conclusion

We present a new C/C++ program defect benchmark, BugsC++, for the evaluation of automatic debugging research. BugsC++ contains 209 real-world bugs collected from 22 projects that are written in C/C++. It provides CLIs to checkout, build, and test individual bugs in a Docker environment that ensures reproducibility. We will endeavor to extend BugsC++ by adding new bugs and projects, as well as improve the usability of the benchmark.

Acknowledgment

All authors have been supported by Institute for Information & communications Technology Promotion grant funded by the Korean government (MSIT) (No.2021-0-01001).

References

[1] B. Hailpern and P. Santhanam, “Software debugging, testing, and verification,” *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[2] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Transactions on Software Engineering*, vol. 42, no. 8, p. 707, August 2016.

[3] L. Gazzola, D. Micucci, and L. Mariani, “Automatic software repair: A survey,” *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[4] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for java programs,” in *ISSTA*, 2014, pp. 437–440.

[5] H. Do, S. G. Elbaum, and G. Rothermel, “Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.” *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[6] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, “Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria,” in *ICSE 1994*, May 1994, pp. 191–200.

[7] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, “Repairing crashes in android apps,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 187–198.

[8] P. Orvalho, M. Janota, and V. Manquinho, “C-Pack of IPAs: A C90 program benchmark of introductory programming assignments,” 2022.

[9] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury, “Codeflaws: a programming competition benchmark for evaluating automated program repair tools,” in *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 180–182.

[10] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, “Where is the bug and how is it fixed? an experiment with practitioners,” in *ESEC/FSE*, 2017, pp. 1–11.

[11] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, “A feasibility study of using automated program repair for introductory programming assignments,” in *ESEC/FSE*, 2017, pp. 740–751.

[12] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, “The manybugs and introclass benchmarks for automated repair of c programs,” *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[13] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, “Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies,” in *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*, 2019.

[14] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. 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*, 2018, pp. 10–13.

[15] Q.-C. Bui, R. Scandariato, and N. E. D. Ferreyra, “Vul4J: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques,” in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 464–468.

[16] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc, and A. Mesbah, “Bugsjs: a benchmark of javascript bugs,” in *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification*, 2019, pp. 90–101.

[17] V. Csuvik and L. Vidács, “Fixjs: A dataset of bug-fixing javascript commits,” in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 712–716.

[18] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, and Q. e. a. Tay, “BugsInPy: A database of existing bugs in python programs to enable controlled testing and debugging studies,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1556–1560.

[19] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury, “Refactoring based program repair applied to programming assignments,” in *Proceedings of the 34th International Conference on Automated Software Engineering*, 2019, pp. 388–398.

[20] S. Benton, A. Ghanbari, and L. Zhang, “Defexts: A curated dataset of reproducible real-world bugs for modern jvm languages,” in *Proceedings of the 41st International Conference on Software Engineering Companion*, 2019, pp. 47–50.

[21] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, “BugSwarm: Mining and continuously growing a dataset of reproducible failures and fixes,” in *Proceedings of the 41st International Conference on Software Engineering*, 2019, pp. 339–349.

[22] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “QuixBugs: A multilingual 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*, 2017, pp. 55–56.

[23] M. Böhme and A. Roychoudhury, “Corebench: Studying complexity of regression errors,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 105–115.

[24] C. Le Goues, S. Forrest, and W. Weimer, “Current challenges in automatic software repair,” *Software Quality Journal*, vol. 21, no. 3, pp. 421–443, 2013.