Poster: BugZoo – A Platform for Studying Software Bugs

Christopher Steven Timperley
Carnegie Mellon University
ctimperley@cs.cmu.edu

Susan Stepney
University of York
susan.stepney@york.ac.uk

Claire Le Goues
Carnegie Mellon University
clegoues@cs.cmu.edu

Abstract

Proposing a new method for automatically detecting, localising, or repairing software faults requires a fair, reproducible evaluation of the effectiveness of the method relative to existing alternatives. Measuring effectiveness requires both an indicative set of bugs, and a mechanism for reliably reproducing and interacting with those bugs. We present BugZoo: a decentralised platform for distributing, reproducing, and interacting with historical software bugs. BugZoo connects existing datasets and tools to developers and researchers, and provides a controlled environment for conducting experiments. To ensure reproducibility, extensibility, and usability, BugZoo uses Docker containers to package, deliver, and interact with bugs and tools. Adding BugZoo support to existing datasets and tools is simple and non-invasive, requiring only a small number of supplementary files. BugZoo is open-source and available to download at: https://github.com/squaresLab/BugZoo

ACM Reference format:
Christopher Steven Timperley, Susan Stepney, and Claire Le Goues. 2018. Poster: BugZoo – A Platform for Studying Software Bugs. In *Proceedings of International Conference on Software Engineering, 2018, Gothenburg, Sweden, May 27–June 3, 2018 (ICSE '18 Companion),* 2 pages.
DOI: 10.1145/3183440.3195050

1 Introduction

Empirical evaluations are a fundamental part of software engineering research. Introducing a new technique for solving a particular problem (e.g., program repair, fault localisation, test generation) entails measuring its effectiveness on a common benchmark and comparing its results to those obtained by similar techniques. Benchmarks for these types of studies typically consist of a dataset of software versions that we shall refer to as *software snapshots*. For many techniques (e.g., program repair or fault localization), snapshots correspond to known bugs [3, 5, 7]; for others (e.g., genetic improvement [6] or decompilation [4]), simply a fixed release. We focus our attention on snapshots representing bugs, but our framework generalises.

To conduct a high-quality empirical study, one requires both a suitable dataset of software snapshots, and a controlled environment in which to reproducibly interact with them. This environment must possess three qualities:

1. *Reproducibility:* Program behaviour is often dependent upon the exact configuration of its host environment at both compile- and run-time (e.g., library and compiler versions, environment variables). Snapshot behavior should be reliably immutable, allowing results to be accurately and independently reproduced. Simply providing a set of natural-language instructions for building and using snapshots does not ensure this reproducibility. Ideally, tests used by the program in the snapshot should not be flaky.
2. *Extensibility:* Others should be able to reproducibly evaluate new techniques on the datasets used in previous evaluations, and reliably extend those datasets.
3. *Usability:* Environments should be lightweight, both in terms of disk space and their run-time overhead, and should be easy to install and use.

The current best practice [5] uses a monolithic virtual machine (VM) to provide an environment in which to interact with snapshots. This approach ensures reproducibility, but does so at the cost of extensibility and usability. The need to compile tools and techniques using the same set of libraries and binaries used by the snapshots (which may rely on old technology) quickly leads to "DLL hell": the environment required by a tool may conflict with the environment provided by the VM. Modifying the VM to overcome this problem compromises reproducibility, as snapshots are no longer guaranteed to behave as they originally did. VMs also incur a significant penalty to performance, and require considerable disk space.

2 BugZoo

We present BugZoo, a decentralised platform for distributing, reproducing, and interacting with historical bugs, connecting datasets and tools, and providing a controlled environment for conducting experiments. Developers and curators can connect their existing tools and datasets to the BugZoo platform by simply adding a set of supplementary

ICSE '18 Companion, Gothenburg, Sweden

 978-1-4503-5663-3/18/05…$15.00
DOI: 10.1145/3183440.3195050

YAML files, known as *manifest files*, to the Git repository that hosts their tool or dataset. Manifest files identify the tool or dataset, and provide machine-readable instructions for building and interacting with the bug or tool. BugZoo thus allows users to independently add their tools and datasets using a small number of non-invasive supplementary files. Users maintain ownership and control over their artifacts, and the decentralised system avoids the bottlenecks and scalability issues common with a centralised approach.

BugZoo uses instructions in the manifest files to build Docker[1] containers that package and deliver snapshots and tools. Docker container images provide a lightweight means of encapsulating a piece of software and all of its dependencies into a single, portable executable package, known as a container.[2] Containers virtualise at the operating-system (rather than hardware-) level by sharing the kernel of the host machine, allowing them to use fewer compute and memory resources than their VM counterparts [2]. Furthermore, since Docker containers are built as a series of file system differences (known as *layers*), they typically size in the tens of MBs. BugZoo thus avoids the problem of conflicting dependencies by ensuring that each bug and tool provides an isolated execution environment (container).

BugZoo provides a range of interfaces, suited to different use cases. The command-line interface allows users to easily register, download, and build bugs and tools provided by remote Git repositories, and to quickly provision interactive containers for studying the bug and running experiments. The RESTlike API and its associated Python bindings allow developers to safely perform structured interactions (e.g., applying patches, executing specific tests, collecting coverage) with historical bugs from within their own programs. BugZoo can also be used as a library by dynamic program analysis tools (e.g., program repair, fault localisation, test generation) to provide reliable and reproducible results, and to reduce code repetition.

3 Related Work

The Software-artifact Infrastructure Repository (SIR) [1], introduced in 2005, contains over 100 versions of 85 programs, written in C, C++, C#, and Java. Unlike later datasets, such as ManyBugs and Defects4J [3], SIR is almost entirely composed of hand-seeded, artificial faults. Furthermore, SIR does not provide a controlled environment for studying its subjects, such as a VM or a container, hampering reproducibility.

ManyBugs [5] is a collection of 185 historical bugs from several real-world C programs, including PHP and Python, and has been used extensively for studies on program repair. The majority of the bugs within the ManyBugs dataset are supported by BugZoo.

Defects4J [3] is a more recently created repository of software faults, introduced in 2014, containing over 300 real-world bugs in several large-scale Java projects. Like BugZoo, Defects4J also provides a CLI and a Java API for interacting with its bugs. Due to the nature of its design, Defects4J only supports bugs in Java programs (that must be compiled using a specific version of Java). BugZoo is not tied any particular language and offers a more general solution.

Unlike the SIR and ManyBugs, BugZoo does not provide a dataset of faulty programs. Rather, it provides a decentralised platform for sharing, persisting, and interacting with both tools and datasets.

4 Conclusion

We have presented BugZoo as a platform for connecting empirical software engineering researchers to software snapshots, allowing them to conduct high-quality, reproducible experiments and evaluations. BugZoo can be used in studies of program repair, fault localisation, automated test generation, specification mining, and genetic improvement.

To learn more about BugZoo, see:

https://github.com/squaresLab/BugZoo.

5 Acknowledgements

This research was partially funded by AFRL (#FA8750-15-2-0075) and DARPA (#FA8750-16-2-0042), and an EPSRC DTG; the authors are grateful for their support. Any opinions, findings, or recommendations expressed are those of the authors and do not necessarily reflect those of the US Government.

References

[1] H Do, S Elbaum, and G Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering* 10, 4 (1 Oct. 2005), 405–435.

[2] W Felter, A Ferreira, R Rajamony, and J Rubio. 2015. An updated performance comparison of virtual machines and Linux containers. In *International Symposium on Performance Analysis of Systems and Software (ISPASS '15)*. 171–172.

[3] R Just, D Jalali, and M 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 (ISSTA '14)*. ACM, New York, NY, USA, 437–440.

[4] J Lacomis, A Jaffe, E J Schwartz, C Le Goues, and B Vasilescu. 2018. Statistical Machine Translation is a Natural Fit for Identifier Renaming in Software Source Code. In *Statistical Modeling of Natural Software Corpora, 2018 AAAI Workshop*. (To appear).

[5] C Le Goues, N Holtschulte, E K Smith, Y Brun, P Devanbu, S Forrest, and W Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *Transactions on Software Engineering* 41, 12 (Dec. 2015), 1236–1256.

[6] J Petke, M Harman, W B Langdon, and W Westley. 2014. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In *European Conference on Genetic Programming*. Springer, Berlin, Heidelberg, 137–149.

[7] C S Timperley, S Stepney, and C Le Goues. 2017. An Investigation into the Use of Mutation Analysis for Automated Program Repair. In *Symposium on Search-Based Software Engineering (SSBSE '17)*. 99–114.

[1] https://www.docker.com. [Accessed November, 2017.]

[2] https://www.docker.com/what-container [Accessed June, 2017.]