Bugs.jar: A Large-scale, Diverse Dataset of Real-world Java Bugs

Ripon K. Saha* Yingjun Lyu† Wing Lam‡ Hiroaki Yoshida* Mukul R. Prasad*
*Software Quality and Security Lab, Fujitsu Laboratories of America, Inc., USA
†Department of Computer Science, University of Southern California, USA
‡Department of Computer Science, University of Illinois at Urbana-Champaign, USA

ABSTRACT

We present Bugs.jar, a large-scale dataset for research in automated debugging, patching, and testing of Java programs. Bugs.jar is comprised of 1,158 bugs and patches, drawn from 8 large, popular open-source Java projects, spanning 8 diverse and prominent application categories. It is an order of magnitude larger than Defects4J, the only other dataset in its class. We discuss the methodology used for constructing Bugs.jar, the representation of the dataset, several use-cases, and an illustration of three of the use-cases through the application of 3 specific tools on Bugs.jar, namely our own tool, Elixir, and two third-party tools, Ekstazi and JaCoCo.

KEYWORDS

Reproducible Bugs, Large-Scale Dataset, Java Programs

1 INTRODUCTION

Software applications pervade every aspect of commercial and consumer life today. The meteoric growth of software over the past decade has brought renewed focus on the reliability and quality of software applications, and indeed the significant costs currently incurred in assuring this reliability [2]. Research on automated debugging, patching and testing of software promises to substantially lower these costs. However, there is always a need for realistic, large-scale datasets to drive the research towards practical solutions. This paper makes a contribution in this area.

Java continues to be one of the leading programming languages today, according to the TIOBE Index [10]. Research on automated debugging and patching of C programs has benefitted immensely from the several realistic datasets of C program bugs currently available, including ManyBugs [4], CoreBench [1], and IntroClass [4]. The Defects4J dataset [5] has played a similar role for Java programs but continues to be the only representative in its class. Further, the subject systems in Defects4J are not diverse enough to represent the entire gamut of current Java applications, and by implication, the diversity of Java bugs. For instance, 4 out of the 6 Defects4J systems (Commons Lang, Commons Math, Joda-Time, and JfreeChart) are actually libraries. For both the above reasons, a second bug dataset, particularly one spanning popular application categories *not* covered in Defects4J, could be a valuable asset for research in automated debugging, patching and testing of Java programs.

In service of the above objective we have developed Bugs.jar, a new large-scale, diverse dataset of 1,158 real bugs and patches from 8 large, popular open-source Java projects, spanning 8 distinct and prominent Java application domains. Bugs.jar is publicly available at https://github.com/bugs-dot-jar/bugs-dot-jar. This paper describes the methodology used for constructing Bugs.jar, the representation of the dataset, specific use-cases, and illustration of three use-cases by using Bugs.jar with three specific tools, namely our own tool, Elixir [9], and two third-party tools, Ekstazi [3] (http://ekstazi.org/) and JaCoCo (http://www.jacoco.org/jacoco/).

2 DATASET CONSTRUCTION

2.1 Objectives

Our primary objective in constructing Bugs.jar was to organize a collection of bugs and corresponding patches, from a diverse set of large real-world Java software, that could serve as a benchmark suite for research in automated debugging, patching, and testing of Java programs. Specifically, for each bug we require (1) the buggy version of the source code, (2) a bug report describing the nature of the bug, (3) a test-suite, serving as a correctness specification, comprising at least one failing (fault-revealing) test case and one passing test case (to guard against regression), and (4) the developer's patch to fix the bug, which passes *all* the test cases. The design of Bugs.jar was driven by the following four broad criteria.

1. Real-world relevance: The goal was to focus on large, active projects, with a rich development history. This would allow for evaluating software engineering techniques at scale. It would also provide a vast and diverse database of bugs.

2. Diversity: The chosen software projects should cover the spectrum of applications typically implemented using Java, allowing for an even richer diversity of bugs and patches.

3. Reproducibility: Many automated debugging and repair techniques, that could benefit from such a bug dataset, rely on test-suite executions to give consistent results over repeated executions. Thus, test-cases containing sources of randomness, or indeed bug instances with such test cases, cannot be included.

4. Automatability: Maximizing automation is crucial to the development of a large-scale dataset. It also limits errors and subjective bias introduced by manual examination. Therefore we limited our search to projects and project ecosystems that followed rigorous development practices, including having comprehensive, well-maintained test-cases, a dedicated and actively maintained bug tracking system, and descriptive commit logs to facilitate identification of bug-fixing commits. We also enforced uniformity in

MSR 2018, 28–29 May, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196473

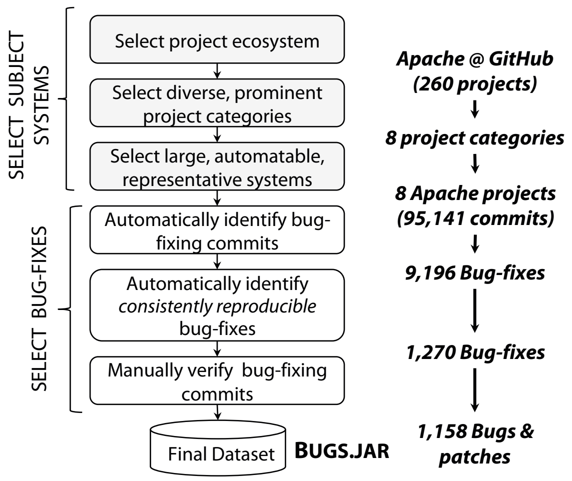
SELECT SUBJECT SYSTEMS
Select project ecosystem
Select diverse, prominent project categories
Select large, automatable, representative systems
SELECT BUG-FIXES
Automatically identify bug-fixing commits
Automatically identify *consistently reproducible* bug-fixes
Manually verify bug-fixing commits
Final Dataset
BUGS.JAR
Apache @ GitHub (260 projects)
8 project categories
8 Apache projects (95,141 commits)
9,196 Bug-fixes
1,270 Bug-fixes
1,158 Bugs & patches

Figure 1: Methodology for creation of dataset

the build and test-authoring frameworks used by the different subject systems. This can prove to be an important issue for program software engineering tools trying to utilize such a dataset [7].

Figure 1 presents an overview of the methodology used to create Bugs.jar. Broadly, it involves first identifying a set of suitable subject systems (projects) and then mining the source repositories of each of these projects for viable bug-patch instances.

2.2 Methodology for choosing projects

After conducting a rigorous search on GitHub and Google Code, we found that projects developed by the Apache Foundation, hosted on GitHub, fulfill our *real-world relevance* and *automatability* criteria. Further, there are several hundred projects in this ecosystem and projects are tagged with one or more of 28 different keywords, such as *library*, *big data*, *network-server*, *etc.,* representing the application domain of the project. This provides an objective path for us to satisfy our *diversity* criterion as well (as discussed below). Thus we chose the Apache ecosystem on GitHub, which has 260 such tagged projects, for constructing Bugs.jar.

First, we grouped the 260 projects, each of which is tagged with one or more of 28 different tags, by tag categories. Note that a project can appear in multiple groups. We then selected the top 8 groups, each of which have 20 or more projects. This strategy respects our *diversity* criterion. Next we selected representative projects from each of the 8 groups that satisfied the following specific criteria: (1) at least 50KLoc and 5,000 commits (*real-world relevance* criterion), and (2) use maven as the underlying build system and JUnit test cases (*automatability* criterion). We selected subjects among the most active projects in each group (measured by the number of commits), provided it passed the above criteria. Further, we chose subjects to reflect the proportion of projects in each category. Table 1 lists the set of chosen subject systems.

2.3 Methodology for filtering bugs

The objective of this step was to select those commits that purely correspond to bug fixes, rather than feature enhancements or anything else, and for which the bug could be reproduced reliably.

Identifying bug-fixing commits. All Apache projects use the Jira issue-tracking system. Each issue in Jira is tagged with a *type*, *e.g.,* Bug, Improvement, Task *etc.,* , and a unique alphanumeric issue ID. Further, when (Apache) developers make changes to the source code to resolve a given issue, they generally include the issue ID in the commit message. To identify all bug-fixing commits of a given project we iterate through the complete commit history of the project, searching for issue IDs in commit messages. Any issue ID found in a commit message is checked against the Jira repository for its type, and if it is of type *Bug*, the corresponding commit is selected as a bug-fixing commit. The set of bug-fixing commits selected in this fashion are the input to the subsequent steps.

Consistently reproducible bug-fixes. Once we get the bug fixing commit from the previous step, we extract the version, which we call V_{fix}. Then we run all the test cases on V_{fix}. Ideally, all the test cases should pass on V_{fix}. However, in real-world, large software projects, there may be some broken test cases in practice. If we get any failing test cases, we exclude them from analysis since they are most likely irrelevant to the bug under investigation. Then we apply the bug fixing reverse patch to get buggy version of the source code. We call this version as V_{buggy}. We run the test cases on V_{buggy}. If there is any fault-reproducing test cases, we consider the bug for further investigation. There may be also some flaky test cases, whose results are non-deterministic [6]. These flaky test cases make some bugs reproducible some times but not reproducible in other times. Therefore, we run the previous step 10 times to make sure that the bugs are consistently reproducible.

Manual verification. Finally, when we get a set of consistently reproducible bugs, we manually analyze each bug to make sure that they are indeed a bug, *i.e.,* they are not misclassified as a Bug in the Jira repository. To this end, two authors of this paper independently read each bug report and decided whether a given issue is a bug. In case of any disagreement, we had a group discussion among all the authors to resolve the issue, although such instances were rare. The right-most column of Table 1 lists the final number of bugs included in the dataset, for each subject system.

2.4 Statistics of final dataset

As per Table 1, Bugs.jar consists of 1,158 bugs from 8 subject systems. It is roughly an order of magnitude larger than the Defects4J dataset [5], with 2.93X more bugs (1,158 vs. 395 for Defects4J) and 3.46X larger subject systems (1,167 KLoc of source code for the latest version of the subjects, vs. 337 KLoC for Defect4J), *i.e.,* $2.93 \times 3.46 = 10.14X$ larger. More importantly, it contains applications from 8 different, prominent application categories, while 4 of the 6 subjects in Defects4J (Commons Lang, Commons Math, Joda-Time, and JfreeChart) are actually libraries. Apache Commons Math is the only subject system common to both datasets.

3 DATA REPRESENTATION

Each bug instance in Bugs.jar contains four artifacts: (1) the buggy version of source code, (2) the bug report, (3) the bug reproducing test-suite, and (4) the developer's patch to fix the bug. In the simplest form, Bugs.jar may be bundled with the snapshots of all buggy versions, and distributed as an archive. However, that is certainly not space-efficient for large subjects. Furthermore, such archival

Table 1: Selected subject systems

Project	Purpose	Tags	Commits	Bug Reports	Size (excl. tests) [KLoC]	Total Size [KLoC]	**Bugs Selected**
Accumulo	sorted, distributed key-value store	database	8,714	2,041	371	458	**98**
Camel	routing and mediation engine	network-client, network-server	24,096	1,081	122	257	**147**
Commons Math	math & statistics library	library	5,994	635	93	187	**147**
Flink	streaming dataflow engine	big data	8,906	2,070	171	345	**70**
Jackrabbit Oak	content management system	XML, network-server, library	10,810	1,686	139	228	**278**
Log4J2	logging framework	library	6,971	784	63	104	**81**
Maven	project management	build management	10,264	2,863	81	100	**48**
Wicket	server-side Web app framework	Web framework	19,386	3,770	127	177	**289**
Total			95,141	14,930	1,167	1,856	**1,158**

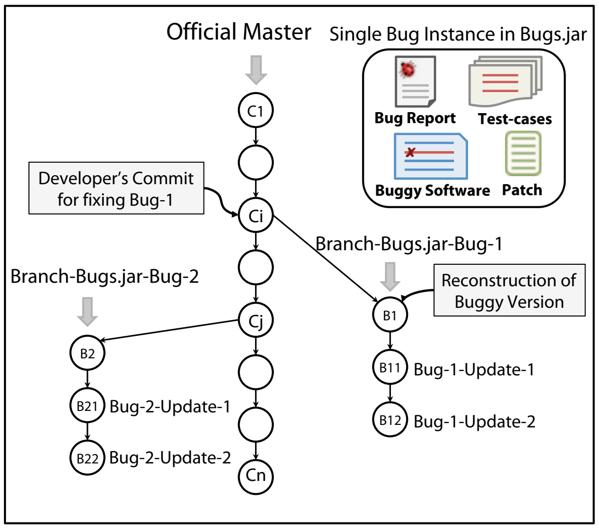

Figure 2: Bugs.jar Representation on GitHub

of bug instances lacks transparency, and flexibility to extend the dataset. Therefore, we carefully designed a tree data structure for Bugs.jar and stored it in a GitHub repository linking Bugs.jar subjects to their respective official repositories. We leveraged several Git features to represent Bugs.jar that promotes the transparency and flexibility to update and extend it, while being space efficient.

Let us assume that for a given subject project (*e.g.,* Apache Camel), there are n commits ranging from C_1 to C_n. Recalling from Section 2.3, each bug instance in Bugs.jar is reconstructed from its corresponding fixed version V_{fix}. Let us assume that C_i is such a bug fixing commit in Figure 2. We make a branch with a Bugs.jar ID (*e.g.,* Bugs.jar-Bug-1), and we commit the reconstructed buggy version V_{buggy}, presented as node $B1$. We also added all the relevant information such as the bug report, test results, and so on in a hidden directory so that they do not affect the project's purity. [1] Since the reconstruction of a buggy version may involve not only reverting the bug fixing changes but also changes in the build scripts or filtering out refactoring related changes, this design enables a user to see the exact changes during the reconstruction of the buggy version with respect to the original fixed version. In our experience of evaluating Elixir on Defects4J, although we frequently needed the change information for a given buggy version, there was no convenient way to extract that since each bug instance in Defects4J is completely isolated from the official subject repository.

Another important reason to choose this data structure is that Bugs.jar also may undergo changes due to its own bug fixes or feature improvements. Under the present design, we can keep updating the dataset by making further commits to the specific branch for the bug ID (*e.g.,* $B11$). Therefore, the updating process is transparent to the user. To extend Bugs.jar with new bugs, we simply start another branch at an appropriate position (*e.g.,* C_j). Bugs.jar maintains a separate Git repository on GitHub for each subject which is linked to the official repository of that subject. All the subject Git repositories are grouped together in a parent Git repository. Due to the novel data representation of Bugs.jar, it cannot be simply merged to any existing bug dataset such as Defects4J.

4 USE CASES

4.1 Research Opportunities

Bugs.jar can be a valuable resource for driving research in the broad area of automated debugging, patching, and testing of Java programs. More specifically, it can be used for designing innovative techniques related to code coverage, mutation testing, test selection, prioritization, and reduction, test case generation, bug localization, learning invariants, anomaly detection, automatic program repair, and so on. Furthermore, since each bug in Bugs.jar comes with a bug report, it would be a valuable asset for research areas such as information retrieval based bug localization, bug report triaging, bug report quality analysis, program comprehension, etc.

4.2 Using Bugs.jar

Using Bugs.jar is simple. Users can download the entire copy of Bugs.jar in a directory of a local or remote computer by simply cloning the Bugs.jar repository from GitHub. Then, for a given subject project, users can check out a specific branch using the corresponding Bugs.jar bug ID. This will move the Git pointer to the latest version of Bugs.jar instance for that bug ID that contains the buggy code, the test suite with bug reproducing test cases, and properly configured build and test scripts (*e.g.,* POM files for maven build systems). This snapshot ensures that if there exist any broken test cases that are irrelevant to the bug, they will not be run during testing. Also, there is a hidden directory called `.bugs-dot-jar` that contains useful information regarding the developer's patch, and

[1]Many modern projects do not build if a foreign file is detected in the codebase.

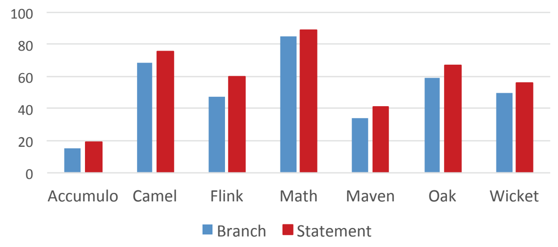

Figure 3: Statistics of Code Coverage by JaCoCo

Table 2: Median Proportion of Test Cases Selected By Ekstazi

Accumulo	Camel	Flink	Math	Oak	Maven	Wicket
23%	1%	17%	16%	33%	22%	8%

detailed test results that we generated during the construction of Bugs.jar. A more comprehensive manual on the usage of Bugs.jar is provided in the Bugs.jar system as a `README.md` file.

4.3 Our Experience of Using Bugs.jar

To demonstrate that Bugs.jar is easily usable by various external and internal tools, this section describes our experiences of applying three different tools on Bugs.jar for three different applications. More specifically, we applied two external tools JaCoCo and Ekstazi on Bugs.jar for code coverage collection and regression test selection, respectively. We also ran our own automatic program repair tool, Elixir on Bugs.jar. Since all these experiments are expensive, we performed them on the 252 one-hunk bugs of Bugs.jar.

Collection of Code Coverage Using JaCoCo. Standard code coverage information is collected and used in many dynamic techniques including fault localization, and test suite evaluation. JaCoCo is a popular and publicly available tool for obtaining code coverage for a given Java project. In order to run JaCoCo, we included the JaCoCo maven plugin in the build of each bug instance. Figure 3 shows that the average code coverage across all the experimented buggy versions of each system. The statement (branch) coverage ranges from 19% (15%) for Accumulo to 89% (85%) for Commons Math. Therefore, Bugs.jar is diverse in terms of test coverage. Furthermore, the successful run of JaCoCo on Bugs.jar demonstrates that Bugs.jar can be used in analyses requiring code coverage.

Regression Test Selection (RTS) Using Ekstazi. RTS is a well known technique to select the relevant test cases for any given changes in a project. Ekstazi is an industry-standard publicly available RTS tool. Like JaCoCo, we were also able to run Ekstazi successfully on Bugs.jar by including its maven plug-in in the build system of each bug instance. Table 2 presents the median proportion of selected tests across all the experimented buggy versions of each system. Ekstazi did not generate results for around 5% of the bugs.

Automatic Program Repair Using Elixir. Running an automatic repair tool is challenging for any dataset since it involves both static and dynamic program analyses. Furthermore, various third-party tools and libraries enable such sophisticated analysis. For example, Elixir uses the ASM byte code library to instrument programs' source code, Spoon library [8] to modify a program at the abstract syntax tree (AST) level, `javax.tools` for in-memory compilation, and JUnit APIs to run the test cases programmatically.

Table 3: Patch Generation Summary by Elixir

Project	# One-hunk Bugs	Correct	Incorrect
Accumulo	21	1	0
Camel	31	4	3
Commons Math	41	10	4
Flink	14	2	0
Jackrabbit Oak	61	4	7
Maven	10	0	0
Wicket	74	11	11
Total	252	32	25

A summary of the results is presented in Table 3. This demonstrates that Bugs.jar supports complex static and dynamic analysis and publicly available libraries that implement these techniques.

Challenges. Running various tools on large systems can be difficult. Although the uniform and efficient architecture of Bugs.jar simplified the process, there are still some challenges depending on the given application. For automatic program repair, we needed to run test cases using JUnit APIs. However, using such APIs was particularly tricky for subjects with many dependent libraries since correctly specifying configurations, such as classpaths was considerably more difficult with the APIs. Also, instrumentation was an issue for a subject such as Log4J2 due to library conflicts.

5 CONCLUSION

In this paper we described Bugs.jar, a large-scale, diverse dataset of real-world Java bugs, aggregated by us. Bugs.jar is comprised of 1,158 bugs and patches, drawn from 8 large, popular open-source Java projects, spanning 8 diverse and prominent application categories. We discussed the methodology used for constructing Bugs.jar, its representation, and illustrated three of its many use-cases through the application of three specific tools on Bugs.jar. We strongly believe that Bugs.jar can be a valuable asset for driving research in automated debugging, patching, and testing of Java programs, and have already made Bugs.jar available to the research community with this sincere hope.

REFERENCES

[1] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: Studying Complexity of Regression Errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 11.
[2] Cambridge University. 2013. Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year. http://www.prweb.com/releases/2013/1/prweb10298185.htm. (2013).
[3] M. Gligoric, L. Eloussi, and D. Marinov. 2015. Ekstazi: Lightweight Test Selection. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*.
[4] 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. *IEEE Transactions on Software Engineering* 41, 12 (Dec 2015).
[5] 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*. ACM.
[6] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM.
[7] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2016. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering* (2016), 1–29.
[8] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* (2015), na.
[9] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. ELIXIR: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 648–659.
[10] TIOBE Index. 2018. http://www.tiobe.com/tiobe-index/. (January 2018).