RegMiner: Mining Replicable Regression Dataset from Code Repositories

Xuezhi Song
Fudan University
China
songxuezhi@fudan.edu.cn

Yun Lin∗
Shanghai Jiao Tong University
China
llmhyy@gmail.com

Yijian Wu∗
Fudan University
China
wuyijian@fudan.edu.cn

Yifan Zhang
National University of Singapore
Singapore
zyifan828@gmail.com

Siang Hwee Ng
National University of Singapore
Singapore
sianghwee@u.nus.edu

Xin Peng
Fudan University
China
pengxin@fudan.edu.cn

Jin Song Dong
National University of Singapore
Singapore
dcsdjs@nus.edu.sg

Hong Mei
Peking University
China
meih@pku.edu.cn

Corresponding author

ABSTRACT
In this work, we introduce a tool, RegMiner, to automate the process of collecting replicable regression bugs from a set of Git repositories. In the code commit history, RegMiner searches for regressions where a test can pass a regression-fixing commit, fail a regression-inducing commit, and pass a previous working commit again. Technically, RegMiner (1) identifies potential regression-fixing commits from the code evolution history, (2) migrates the test and its code dependencies in the commit over the history, and (3) minimizes the compilation overhead during the regression search. Our experiments show that RegMiner can successfully collect 1035 regressions over 147 projects in 8 weeks, creating the largest replicable regression dataset within the shortest period, to the best of our knowledge. In addition, our experiments further show that (1) RegMiner can construct the regression dataset with very high precision and acceptable recall, and (2) the constructed regression dataset is of high authenticity and diversity. The source code of RegMiner is available at https://github.com/SongXueZhi/RegMiner, the mined regression dataset is available at https://regminer.github.io/, and the demonstration video is available at https://youtu.be/yzcM9Y4unok.

CCS CONCEPTS
• Software and its engineering → Software evolution; Maintaining software; Software testing and debugging.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ESEC/FSE ’22, November 14–18, 2022, Singapore, Singapore
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3558929

1 INTRODUCTION
Regression bugs, which make a working function fail, often appear in software projects. A typical regression bug can be formalized into an observation that a test case \( t \) can pass a version \( V_{\text{pass}} \) but fail a version \( V_{\text{fail}} \). Researchers have proposed many techniques for regression analysis, specifically,

• Regression testing works, e.g., the detection of whether a new version can fail a working function or what test cases can discover a regression bug [2, 10, 17, 25].

• Regression localization works, e.g., the explanation why the new version \( V_{\text{fail}} \) causes the failure, and the location of the failure inducing changes between \( V_{\text{fail}} \) and \( V_{\text{pass}} \) [12, 13, 16, 26, 34, 39].

• Regression repair works, e.g., the generation of a fixing version from \( V_{\text{pass}} \), and \( V_{\text{fail}} \) to pass the test \( t \) [19, 31].

While the research works for regression bugs are emerging, the community still lacks a scalable benchmark (such as Defects4j [14]) to evaluate them in a systematic way. Our investigation on 14 research works from the year of 1999 to 2021 [1, 5, 6, 15, 17, 21, 31, 34–40] shows that the mean number of evaluated real-world regressions is 16.7 and the median is 12.5. Moreover, different benchmarks are used in different work, making it difficult to compare their performance. A large-scale benchmark can largely mitigate the issue.
However, the process of collecting and labelling a regression is very costly, which requires to (1) prepare a regression-fixing version $V_{\text{fix}}$, a regression-introducing version $V_{\text{fail}}$, and a previous working version $V_{\text{pass}}$, (2) set up at least one test case passing $V_{\text{fix}}$ and failing $V_{\text{fail}}$, and passing $V_{\text{pass}}$, and (3) isolate an environment where the bug can be well replicated.

In this work, we introduce, RegMiner, to automate the collection of runnable regression bugs from the code repository with zero human intervention. RegMiner takes a set of code repositories as input, and isolates a set of regressions with their running and replicable environment as output. Technically, we construct a regression by searching in code repositories for a regression-fixing commit denoted as $rfc$, a regression-introducing commit denoted as $ric$, a working commit (before $ric$) denoted as $wc$, and a test case denoted as $t$ so that $t$ can pass $rfc$ and $wc$, and fail $ric$. To this end, our approach addresses the technical challenges of (1) predicting the commits with larger potential as regression-fixing commits, (2) identifying relevant code changes in a commit to migrate through the code evolution history, (3) adopting the library upgrades with the history, and (4) minimizing the compilation overhead and handling incompilable revisions. More technical details are reported in our research track work (in ISSTA’22) [29].

We implement RegMiner tool for mining regression bugs from Java projects supported by either Maven or Gradle. We evaluate our tool by running the regression dataset is available at https://regminer.github.io/, and the source code of RegMiner is available at https://github.com/SongXueZhi/RegMiner, the mined regression dataset is available at https://regminer.github.io/, and the demonstration video is available at https://youtu.be/yzcM9Y4unok.

2 REGMINER TOOL

2.1 Mining Process Dashboard

Figure 1 shows the dashboard user interface to run the regression-mining process from the Git repositories.

Start, Resume, and Stop of the Mining Process. On the up-right corner of the dashboard, there is a “Start” button allowing the users to start the mining process. In addition, the user can click the “Stop” button to suspend the mining process, and click the “Start” button to resume the process.

Mining Progress. Once the mining process starts, RegMiner visualizes two processes:

- **Progress of Processing Projects**: We show under the “processed projects” section the ratio of the number of the processed projects over that of the total projects.
- **Progress of Processing Bug-fixing Commits**: We show under the “processed bug-fixing commits” section the ratio of the number of the processed bug-fixing commits over that of the total found bug-fixing commits. Note that, RegMiner will extract potential bug-fixing commits from the code repositories, along with their verifying test cases which pass the commit and fail the commit before. The process verifies whether a bug-fixing commit is a regression-fixing commit, by searching for a working commit where the test case can pass.

Timeline. During the mining process, we further allow the users to investigate any of the found regression bugs on how it is found during the search on the commit history. The “Timeline” session shows stepwise search process to locate the working commit. Each point represents a visited commit during the search. The green points represent the commits where the test case can pass, the red points represent the commits where the test case can fail, and the grey points represent the commits which cannot be compiled. The search process starts with the regression-fixing commit, and ends with the working commit. Users can replay the search process by clicking the “Next” and “Previous” buttons.

2.2 Regression Details

Moreover, the user can further investigate the code and its diff information. In addition, we can also run the commits (i.e., regression-fixing commit, regression-introducing commit, and the working commit), and show the runtime results in the console, as shown in Figure 2. More details can be investigated in our website https://regminer.github.io/.

3 TOOL DESIGN

Figure 3 shows an overview of the RegMiner design. RegMiner takes input as a set of code repositories (a.k.a., Git repositories), and generates regressions formalized as a 4-tuple $\text{reg} = \langle \text{test, rfc, ric, wc} \rangle$ where $rfc$ represents a regression-fixing commit, $ric$ represents a regression-introducing commit, $wc$ represents working commit, and $test$ represents a test case passing $rfc$, failing $ric$, and passing $wc$. The RegMiner framework consists of three high-level modules:

- **Repository Parsing Module**: The module parses the Git repositories into a pool of bugs (i.e., candidate regression) for further mining. With a set of Git repositories, a Repository Parsing Module first scans the bug-fixing commits with regression potential. Specially, the Repository Parsing Module reports a set of 3-tuple $\text{bug} = \langle \text{test, bfc, p} \rangle$ where (1) the test case $\text{test}$ can pass a bug-fixing commit $\text{bfc}$ and fail the commit before $\text{bfc}$ (noted as $\text{bfc} - 1$) and (2) the likelihood of $\text{bug}$ turns out to be a regression. Note that the bug-fixing commits with very low regression potential score $p$ are discarded. More details of the probability calculation can be referred in [29].
- **Commit Validation Module**: The module verifies each bug in the candidate bug pool by whether it can find a commit before $\text{bfc} - 1$ where $\text{test}$ can pass. During the search, the module automatically compiles the project¹, migrates the test case $\text{t}$ with a previous commit, and heuristically locates the target commit with a balance of efficiency and completeness. As a result, we construct a regression bug as a 4-tuple $\text{reg} = \langle \text{test, rfc, ric, wc} \rangle$ where $\text{test}$ can pass $\text{rfc}$, fail $\text{ric}$, and pass $\text{wc}$.
- **User Interaction Module**: The module provides an intuitive user interface based on the mined regression bug dataset.

¹We now support Maven and Gradle project.
Critical Change Recommendation: Given a regression-fixing commit or regression-introducing commit, we adopt state-of-the-art delta-debugging technique to recommend their critical changes.

Change Annotation: We further support users to confirm the recommended critical changes and annotate the mis-reported changes.

Bug Search: In addition, we integrate a keyword-based search engine for users to look for their interested regression bugs, e.g., regarding concurrency, socket, IO, etc.

4 EXPERIMENT

We build RegMiner to mine Java regressions, supporting Maven and Gradle projects in its current implementation. We evaluate RegMiner with the following research questions, more details of our experiment are available at our website (https://regminer.github.io/).

- **RQ1 (Close-world Experiment)**: Whether RegMiner can mine regressions from Git repositories accurately and completely?
- **RQ2 (Open-world Experiment)**: Whether RegMiner can continuously mine authentic regressions from real-world Git repositories? How diverse is our constructed regression dataset?
**RQ3 (Ablation Study):** How each component in RegMiner, i.e., regression potential estimation, test dependency migration, and validation effort minimization, contribute to the mining effectiveness and efficiency?

### 3.1 Results (RQ1): Close-world Experiment
We evaluate RegMiner in the close-world experiment regarding precision and recall. In this study, we manually collect 50 regression-fixing commits and 50 non-regression fixing commits from Github repositories for the measurement of precision and recall. To avoid bias, we construct the regression benchmark without any help of RegMiner. Specifically, we search for closed Github issues each of which uniquely mentions a commit as its solution. Then, we filter those issues based on its labels and description. We prioritize the issues with labels as “regression” or “bug”, or with description with the keyword of “regression”. Then, we confirm the real regressions by manually checking the evolution history for regression-inducing commits, and migrating the test cases. As a result, we confirmed 50 regressions from 23 Java projects by this means. By similar means, we identified 50 non-regression bugs.

We apply RegMiner on all the regression and non-regression fixing commits to measure the precision and recall. Let the number of true regressions to be $N$ ($N = 50$), the number of reported regressions to be $M$, and the number of reported true regressions to be $K$, the precision is $\frac{K}{M}$, the recall is $\frac{K}{N}$. Overall, we achieve 100% precision and 56% recall in the experiment.

### 3.2 Results (RQ2): Open-world Experiment
RegMiner is designed to mine regressions from Git repositories with zero human intervention. In this experiment, we collect Git repositories from Github and run RegMiner to mine regressions for 8 weeks. We evaluate RegMiner regarding the authenticity and the diversity of the mined regressions in both quantitative and qualitative manner. Overall, RegMiner constructs a regression dataset consisting of 1035 regressions over the 147 projects within 8 weeks.

### 3.3 Results (RQ3): Ablation Study
In the study, we disable and enable the function of test case migration and validation effort minimization respectively to evaluate their effectiveness. Table 1 shows that the overall regression retrieval performance. Overall, comparing to the baselines, our solutions show their effectiveness.

---

**Figure 3: Design of RegMiner: A producer-consumer architecture to progressively search for the regression bugs. Follow-up analysis is built upon the collected regression bugs such as critical change recommendation, change annotation, and bug search.**

<table>
<thead>
<tr>
<th>Approach</th>
<th>Close-world Experiment</th>
<th>Open-world Experiment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prec</td>
<td>Rec</td>
</tr>
<tr>
<td>RegMiner</td>
<td>1.0</td>
<td>0.56</td>
</tr>
<tr>
<td>RegMiner−TDM</td>
<td>1.0</td>
<td>0.32</td>
</tr>
<tr>
<td>RegMiner−VEA+biset</td>
<td>1.0</td>
<td>0.47</td>
</tr>
</tbody>
</table>

## 5 RELATED WORK
Researchers have constructed many bug datasets in the community. Many datasets are constructed from (1) programming assignments and competitions such as Marmoset [30], QuizBugs [20], IntroClass [18], CodeFlaws [32], etc., (2) open source projects (e.g., SIR [8], DbgBench [4], Defects4J [14], BugsJS [15], Bugs.jar [28], etc.), and (3) runtime continuous integration scenarios (e.g., BEARS [24] and BugSwarm [33]). The most relevant dataset CoREBench [3], which is a regression dataset of 70 C/C++ regressions.

Those manually constructed bug datasets limit the scalability and representativeness of the bugs. Dallmeier and Zimmermann [7] make the first attempt to construct bug dataset in a semi-automatic way. They construct the dataset by linking the bug issues and commits. Zhao et al. [41] replicate bugs based on Android bug reports. Recently, CI (Continuous Integration)-based techniques are emerging. BEARS [24] and BugSwarm [33] construct the bug dataset by collecting the buggy and the patched versions on Continuous Integration systems. In comparison, RegMiner is the first tool to mine regression bugs, with fewer assumptions (i.e., we only require Git repositories supported by Maven or Gradle).

## 6 CONCLUSION
In this work, we introduce RegMiner which can automatically construct a regression dataset. In the future, we will further improve the regression retrieval efficiency such as synthesizing regression tests for enlarging the pool of regression-fixing commits, and designing more accurate migration techniques. Moreover, we will enhance RegMiner by supporting manual correction of recommended critical changes and providing relevant APIs from RegMiner to facilitate a variety of research experiments on software testing, debugging, and repair [9, 19, 21–23, 31, 34, 39].
ACKNOWLEDGMENTS

We sincerely thank anonymous reviewers for their comments to improve this work. This research is supported in part by the Minister of Education, Singapore (TZE20120-0019, T1-251RES1901, MOET32020-0004), A*STAR, CISCO Systems (USA) Pte. Ltd and National University of Singapore under its Cisco-NUS Accelerated Digital Economy Corporate Laboratory (Award I21001E0002), and National Natural Science Foundation of China (62172099).

REFERENCES