RegMiner: Mining Replicable Regression Dataset from Code Repositories

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

Yun Lin*
Shanghai Jiao Tong University
National University of Singapore
China/Singapore
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

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 experients 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**.

*Corresponding author

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

KEYWORDS

mining code repository, bug collection, regression bug

ACM Reference Format:
Xuezhi Song, Yun Lin, Yijian Wu, Yifan Zhang, Siang Hwee Ng, Xin Peng, Jin Song Dong, and Hong Mei. 2022. RegMiner: Mining Replicable Regression Dataset from Code Repositories. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 5 pages. 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_{pass} but fail a version V_{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_{fail} causes the failure, and the location of the failure inducing changes between V_{fail} and V_{pass} [12, 13, 16, 26, 34, 39].
- Regression repair works, e.g., the generation of a fixing version from V_{pass} and V_{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, 27, 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_{fix}, a regression-introducing version V_{fail}, and a previous working version V_{pass}, (2) set up at least one test case passing V_{fix} and failing V_{fail}, and passing V_{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-inducing 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 approach with a close-world and an open-world experiment. In the close-world experiment, RegMiner achieves 100% precision and 56% recall on a benchmark consisting of 50 regression bugs and 50 non-regression bugs. In the open-world experiment, we run RegMiner on 147 code repositories within 8 weeks. RegMiner reports 1035 regressions which construct the largest Java regression dataset to the best of our knowledge. Our ablation study also shows the effectiveness and efficiency of our technical designs. 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-bug 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 $reg = \langle test, rfc, ric, wc \rangle$ 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 $bug = \langle test, bfc, p \rangle$ where (1) the test case $test$ can pass a bug-fixing commit bfc and fail the commit before bfc (noted as $bfc - 1$) and (2) the likelihood of 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 $bfc - 1$ where $test$ can pass. During the search, the module automatically compiles the project[1], migrates the test case t 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 $reg = \langle test, rfc, ric, wc \rangle$ where $test$ can pass rfc, fail ric, and pass wc.
- **User Interaction Module**: The module provides an intuitive user interface based on the minded regression bug dataset.

[1]We now support Maven and Gradle project.

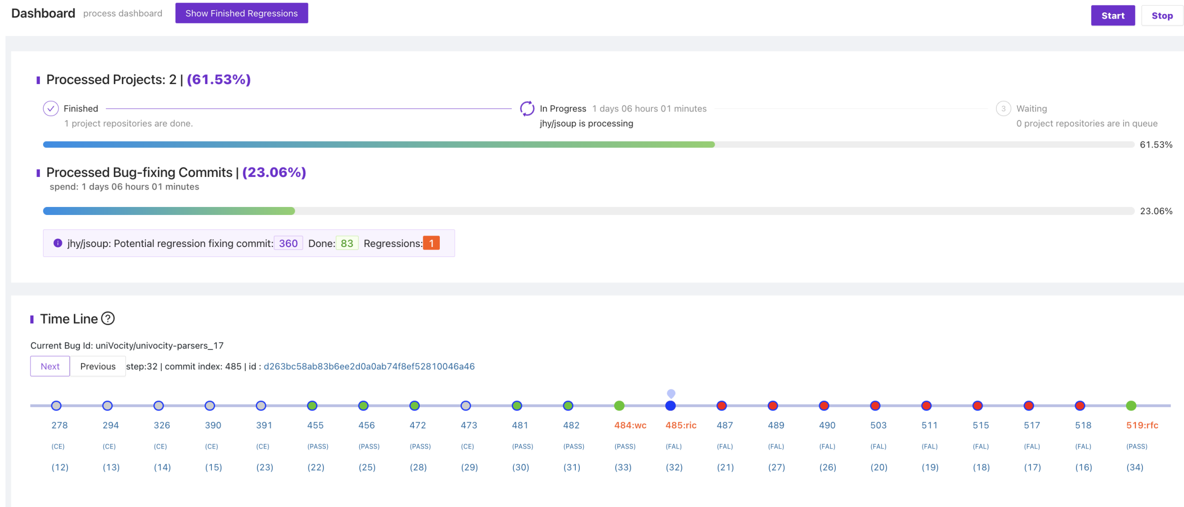

Figure 1: User interface of RegMiner: We show progressive information to mine regression bugs from predefined Git projects.

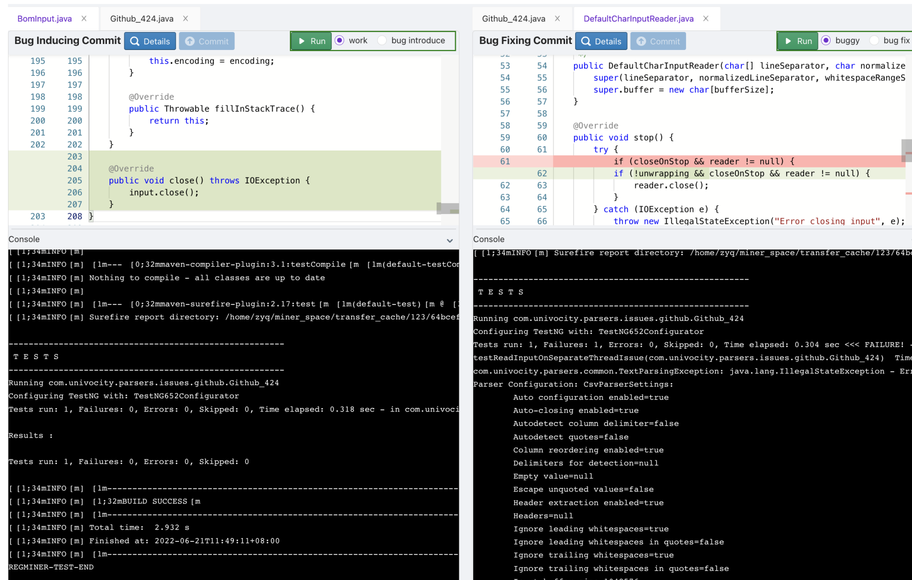

Figure 2: User interface of details of an individual regression bug. User can run the test case against the commits/versions to check their runtime output in the console.

(1) **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.
(2) **Change Annotation**: We further support users to confirm the recommended critical changes and annotate the mis-reported changes.
(3) **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?

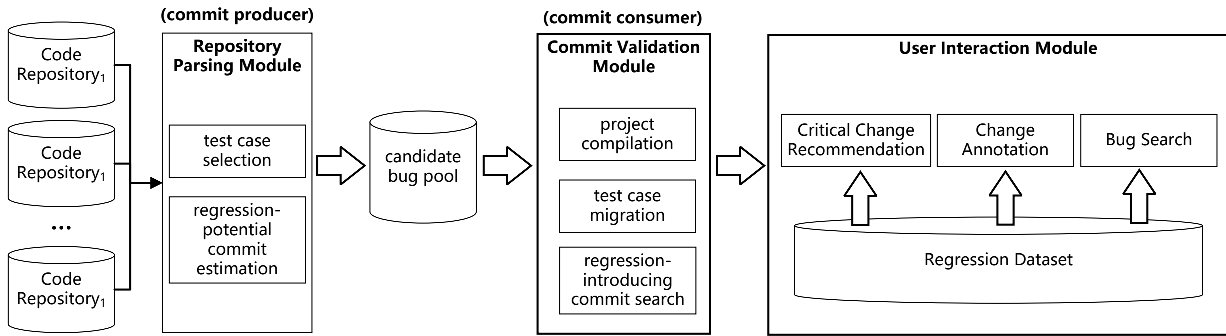

Figure 3: Design of RegMiner: A producer-consumer architecture to progressively search for the regression bugs. Follow-up analysis is built upon the collected regression bugs such as critical change recommendation, change annotation, and bug search.

- **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.

Table 1: The overall performance of regression retrieval

Approach	Close-world Experiment			Open-world Experiment	
	Prec	Rec	Time (h)	#Reg	Time (h)
RegMiner	1.0	0.56	4.29	1035	135.38
RegMiner$_{\neg TDM}$	1.0	0.32	2.21	604	64.29
RegMiner$_{\neg VEM+biset}$	1.0	0.47	2.74	629	73.84

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], QuixBugs [20], IntroClass [18], Codeflaws [32], etc., (2) open source projects (e.g., SIR [8], DbgBench [4], Defects4j [14], BugsJS [11], 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 (T2EP20120-0019, 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

[1] Cyrille Artho. 2011. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer* 13, 3 (2011), 223–246.

[2] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1–12.

[3] Marcel Böhme and Abhik Roychoudhury. 2014. Corebench: Studying complexity of regression errors. In *Proceedings of the 2014 international symposium on software testing and analysis*. 105–115.

[4] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 117–128.

[5] Robert Brummayer and Armin Biere. 2009. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. 1–5.

[6] Dekel Cohen and Amiram Yehudai. 2015. Localization of real world regression Bugs using single execution. *arXiv preprint arXiv:1505.01286* (2015).

[7] Valentin Dallmeier and Thomas Zimmermann. 2007. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 433–436.

[8] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10, 4 (2005), 405–435.

[9] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[10] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential regression testing for REST APIs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 312–323.

[11] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. 2019. BugsJS: a benchmark of JavaScript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 90–101.

[12] Renáta Hodován and Ákos Kiss. 2016. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. 31–37.

[13] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Coarse hierarchical delta debugging. In *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 194–203.

[14] 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.

[15] Alireza Khalilian, Ahmad Baraani-Dastjerdi, and Bahman Zamani. 2021. CGenProg: Adaptation of cartesian genetic programming with migration and opposite guesses for automatic repair of software regression faults. *Expert Systems with Applications* 169 (2021), 114503.

[16] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 2018. HDDr: a recursive variant of the hierarchical delta debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 16–22.

[17] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 821–830.

[18] 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.

[19] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[20] 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.

[21] Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. Graph-based seed object synthesis for search-based unit testing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1068–1080.

[22] Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. 2020. Recovering fitness gradients for interprocedural Boolean flags in search-based testing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 440–451.

[23] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. 2017. Feedback-based debugging. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 393–403.

[24] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 468–478.

[25] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type regression testing to detect breaking changes in Node. js libraries. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[26] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*. 142–151.

[27] Fabrizio Pastore, Leonardo Mariani, and Alberto Goffi. 2013. RADAR: a tool for debugging regression problems in C/C++ software. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1335–1338.

[28] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. 2018. Bugs.jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 10–13.

[29] Xuezhi Song, Yun Lin, Siang Hwee Ng, Yijian Wu, Xin Peng, Jin Song Dong, and Hong Mei. 2022. RegMiner: Towards Constructing a Large Regression Dataset from Code Evolution History. In *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 314—-326.

[30] Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh. 2005. Software repository mining with Marmoset: An automated programming project snapshot and testing system. In *Proceedings of the 2005 international workshop on Mining software repositories*. 1–5.

[31] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated repair of software regressions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 471–482.

[32] 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.

[33] David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 339–349.

[34] Haijun Wang, Yun Lin, Zijiang Yang, Jun Sun, Yang Liu, Jin Song Dong, Qinghua Zheng, and Ting Liu. 2019. Explaining regressions via alignment slicing and mending. *IEEE Transactions on Software Engineering* (2019).

[35] Qiuping Yi, Zijiang Yang, Jian Liu, Chen Zhao, and Chao Wang. 2015. A synergistic analysis method for explaining failed regression tests. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 257–267.

[36] Kai Yu and Mengxiang Lin. 2012. Towards practical debugging for regression faults. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 487–490.

[37] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. 2012. Practical isolation of failure-inducing changes for debugging regression faults. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 20–29.

[38] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. 2012. Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers' perspectives. *Journal of Systems and Software* 85, 10 (2012), 2305–2317.

[39] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software engineering notes* 24, 6 (1999), 253–267.

[40] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2012. Faulttracer: a change impact and regression fault analysis tool for evolving java programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–4.

[41] Yu Zhao, Kye Miller, Tingting Yu, Wei Zheng, and Minchao Pu. 2019. Automatically Extracting Bug Reproducing Steps from Android Bug Reports. In *International Conference on Software and Systems Reuse*. Springer, 100–111.