Poster: Supporting JavaScript Experimentation with BUGSJS

Béla Vancsics*, Péter Gyimesi*, Andrea Stocco†, Davood Mazinanian†, Árpád Beszédes*, Rudolf Ferenc*, Ali Mesbah†

*University of Szeged, Hungary †University of British Columbia, Canada
{vancsics, pgyimesi, beszedes, ferenc}@inf.u-szeged.hu
{astocco, dmazinanian, amesbah}@ece.ubc.ca

***Abstract*—In our recent work, we proposed BUGSJS, a benchmark of several hundred bugs from popular JavaScript server-side programs. In this abstract paper, we report the results of our initial evaluation in adopting BUGSJS to support an experiment in fault localization. First, we describe how BUGSJS facilitated accessing the information required to perform the experiment, namely, test case code, their outcomes, their associated code coverage and related bug information. Second, we illustrate how BUGSJS can be improved to further enable easier application to fault localization research, for instance, by filtering out failing test cases that do not directly contribute to a bug.**

We hope that our preliminary results will foster researchers in using BUGSJS to enable highly-reproducible empirical studies and comparisons of JavaScript analysis and testing tools.

***Keywords*-JavaScript, bug database, real bugs, fault localization, benchmark, reproducibility, BUGSJS.**

I. INTRODUCTION AND MOTIVATION

JavaScript is a popular language for developing highly interactive web applications, recently also adopted for server-side code [1]. However, JavaScript is an error-prone language, due to its asynchronous, dynamic, and loosely typed nature. Research has focused in recent years on devising automated testing techniques for JavaScript, among which automatic fault localization [3], or root cause analysis of bugs [4].

Novel software analysis and testing techniques are typically evaluated through empirical methods (*e.g.*, controlled experiments), which rely on various software-related artifacts, such as source code, test suites, and descriptive bug reports. Comparing the efficacy of these techniques on a common centralized benchmark is imperative to ensure reliability and replicability. Unfortunately, in the JavaScript domain, our survey of the previous studies showed an extensive heterogeneity in the used evaluation subject systems [5].

To fill this gap, we have proposed BUGSJS, a benchmark of several hundreds manually-validated bugs from 10 popular JavaScript programs, along with comprehensive bug reports and one or more test cases that demonstrate the bugs [5]. These artifacts can be used for devising and evaluating analysis and testing techniques for JavaScript, facilitated through an API that allows accessing and executing the faulty and fixed versions of the programs and the corresponding tests. While several benchmarks of bugs have been proposed [6, 7, 8, 9, 11, 12], to our knowledge, BUGSJS is the first benchmark of detailed, descriptive, and curated programs and bug reports for JavaScript.

In this extended abstract, we report the results of using the BUGSJS benchmark and infrastructure to perform an experiment in fault localization, along with some lessons learned and future directions.

II. BUGSJS

BUGSJS includes 453 reproducible bugs from 10 popular Node.js server-side JavaScript programs that adopt the Mocha testing framework. We have mined these projects from GitHub with the following criteria: (i) popularity: stargazers count ≥ 100, (ii) maturity: number of commits > 200, and (iii) recency: year of the latest commit ≥ 2017. Table I shows the characteristics of the selected projects. The last column shows the initial number of bug candidates we considered for inclusion in the benchmark, from which several had to be removed due to reasons explained below.

TABLE I: Subjects included in BUGSJS

	kLOC	Stars	Commits	Forks	Candidate Bugs
Bower	16	15,290	2,706	1,995	10
ESLint	240	12,434	6,615	2,141	559
Express	11	40,407	5,500	7,055	39
Hessian.js	6	104	217	23	17
Hexo	17	23,748	2,545	3,277	24
Karma	12	10,210	2,485	1,531	37
Mongoose	65	17,036	9,770	2,457	56
Node-redis	11	10,349	1,242	1,245	25
Pencilblue	46	1,596	3,675	276	18
Shield	20	6,319	2,036	1,432	10
Total	444	137,493	36,791	21,432	795

The bugs included in BUGSJS are automatically extracted from the issue tracking systems of the included programs. To ensure artefacts traceability, we included *closed issues* which were assigned with a specific *bug* label in the issue tracker.

For each bug, BUGSJS contains the full textual description of the bug reports and their metadata, along with the developers' discussions. Moreover, we have included the corresponding bug-fixing commit in order to keep the source code changes required to fix each bug. All bug-fixing commits in

TABLE II: Manual and dynamic validation statistics

	Bower	ESLint	Express	Hessian.js	Hexo	Karma	Mongoose	Node-redis	Pencilblue	Shield	Total
Candidate bugs	10	559	39	17	24	37	56	25	18	10	795
After manual validation	8	382	33	13	13	26	41	11	8	7	542
After dynamic validation	3	333	27	9	12	22	29	7	7	4	**453**

BUGSJS have been validated and cleaned, both manually—by multiple authors—and dynamically, to ensure that the bugs and their fixes were *relevant*, *isolated*, and *reproducible*.

Relevance means that the bug-fixing commits are actually fixing the bug described in the issue, rather than, for example, implementing a new feature. *Isolation* means that the bug-fixing code exclusively aim at fixing the bug, and no other changes (e.g., refactorings) are interleaved within it. Isolation is particularly important in domains where bug-fixing changes should be clearly identifiable, hence must be *cleaned* (e.g., training machine learning models to learn the fixes from source code to devise automated program repair techniques). Bugs in BUGSJS are also *reproducible*, since we have manually extracted the test cases that demonstrate each bug. Each set of such tests was executed on the buggy revision (i.e., dynamic validation) to ensure their relevance to the bug, and can be leveraged, for instance, to devise novel automated test generation techniques. The results of our validation of the bugs in BUGSJS are reported in Table II.

We have used the GitHub's *fork* feature to include the entire history of the projects in BUGSJS. Each bug in BUGSJS is *tagged* to five source code revisions, as follows:

1) The parent commit of the revision in which the bug was fixed (i.e., the buggy revision);
2) A revision with the original bug-fixing changes (including the production code and the newly added tests);
3) A revision with only the tests introduced in the bug-fixing commit, applied to the buggy revision;
4) A revision only containing the production code changes introduced to fix the bug, applied to the buggy revision;
5) A revision containing both the cleaned fix and the newly added tests, applied to the buggy revision.

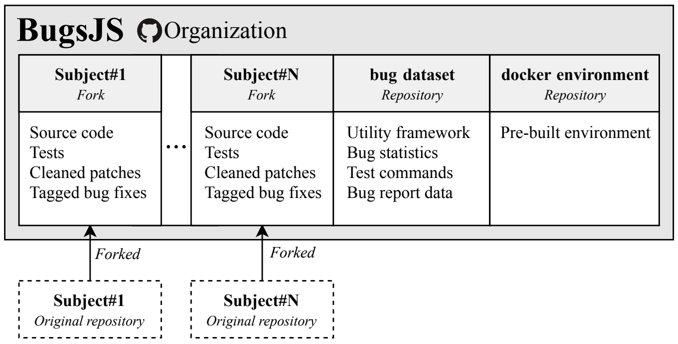

Fig. 1: BUGSJS architecture

BUGSJS consists of two other components (see Figure 1): a `Docker` image that provides a runtime environment, and a bug dataset. The dataset can be queried through an API that supports the following commands: (i) `info`: printing out information about a given bug, (ii) `checkout`: checking-out the source code, (iii) `test`: running all tests and retrieving the overall test suite coverage (iv) `per-test`: running each test individually and retrieving the per-test coverage. These commands facilitate the use of BUGSJS for other researchers to a large extent.

More details about the characteristics of BUGSJS, its architecture, and our design choices can be found in our full paper [5]. The interested reader can find more information on BUGSJS and access the benchmark on our website:

https://bugsjs.github.io/

III. POSSIBLE USE CASES

BUGSJS can be useful to support experimentation in several testing research areas, some of which we list next.

Regression testing. More than 25k test cases included in BUGSJS can facilitate various regression testing studies, e.g., test prioritization, software oracles, or automated test repair.

Bug prediction. The source code and various information pertaining to a large set of bugs available in BUGSJS can be used to construct bug prediction models. The availability of both cleaned and uncleaned bug-fixing patches in the dataset can allow assessing the sensitivity of the proposed models to noise.

Automated program repair. The manually-cleaned patches available in BUGSJS can be used as learning examples for patch generation in novel automated program repair for JavaScript. Also, BUGSJS provides an out-of-the-box solution for automatic dynamic patch validation.

Bug localization. BUGSJS contains pointers to the natural language bug descriptions/discussions for several hundreds of bugs. Devising NLP-based techniques for formulating natural language queries that describe the observed bugs available in BUGSJS is just one sample use case. Also, other approaches to fault localization such as Spectrum-Based Fault Localization can benefit from readily available sets of test cases, detailed code coverage information, test case outcomes and bug positions. We chose this use case as a preliminary example of use of our dataset, which we explain next.

IV. SAMPLE USE CASE: FAULT LOCALIZATION

We used BUGSJS to support an ongoing experimentation on fault localization (FL) [3], a well-established research area that can largely benefit from a dataset of catalogued bugs. To evaluate FL algorithms, we need validated bugs, their fixes, as well as tests to demonstrate the existence of bugs. In our use case, we considered Spectrum-Based Fault Localization (SBFL), a popular FL technique which captures run-time information, such as detailed code coverage, in order to monitor the dynamic behaviour of the software. Then, by associating the test case outcomes with appropriate dynamic information, we can identify the potentially faulty code fragments in the program.

One widely-used program spectra in SBFL is the one which is based on the covered/uncovered statements and methods during test execution. To successfully apply this technique, we require: (i) test coverage information, (ii) tests results, and (iii) modified source code (e.g., statements or methods that were modified during the fixing).

BUGSJS's API provides easy access to all such information. High coverage and non-extreme test-method ratio are also important for the efficiency of FL algorithms. The dataset contains an average of 9327 tests, 1562 methods, 8941 LOC, and features 96% method-level coverage and ≈37:6 test-method ratio.

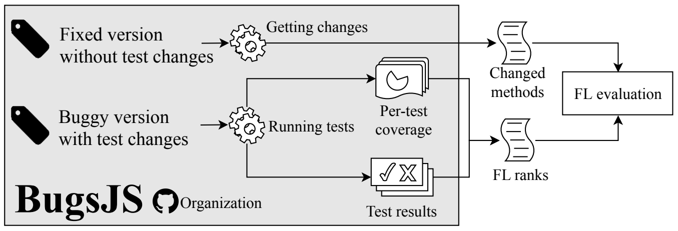

Fig. 2: Fault localization process

The overall FL process using BUGSJS is shown in Figure 2. First, the *per-test* command retrieves which methods are covered by a test, which provides an overview of the full, method-level coverage of the subject program. Second, the *test* command executes all tests and saves the results. The difference between the buggy and the fixed version determines which files were changed by a fix-commit, that is, *the location of the bug* in the source code.

A common approach in FL is to provide a *ranking score* to order program entities (e.g., methods) based on their *likeliness to be buggy.* The ranking score is typically a function of four values for a given method m, all of which can be easily obtained from BUGSJS:

a) m_{ep}: number of passing tests covered by m
b) m_{ef}: number of failing tests covered by m
c) m_{np}: number of passing tests not covered by m
d) m_{nf}: number of failing tests not covered by m

In this work, we used Tarantula [**?**], a popular ranking score defined as follows:

$$Tarantula(m) = \frac{\frac{m_{ef}}{m_{ef}+m_{nf}}}{\frac{m_{ef}}{m_{ef}+m_{nf}} + \frac{m_{ep}}{m_{ep}+m_{np}}}$$

A. Preliminary results: bug-fix patterns and ranks

We applied SBFL with Tarantula technique to the subject program `Hessian.js`, which includes nine (9) bugs.

As part of the analysis and classification of BUGSJS bugs [5], we categorized them based on recurring bug-fix patterns proposed by Pan et al. [15]. By comparing the results of this categorization and the ranking scores of the fixing methods provided by Tarantula, we expect to gather insights on how different types of bugs can be successfully localized.

TABLE III: `Hessian.js` bug-fix patterns and Tarantula ranks

Bug #	Rank	Method	Pattern(s) [15]
5	1	lib/v2/encoder.js:(anonymous_3)	
2	2	lib/v2/decoder.js:(anonymous_15)	IF-RMV
9	2	lib/v1/decoder.js:(anonymous_20)	SQ-RMO
3	3	lib/v1/encoder.js:(anonymous_21)	IF-APCJ
8	3	lib/utils.js:(anonymous_3)	IF-CC
6	4,5	ib/v2/decoder.js:(anonymous_11)	IF-APC
4	7	lib/v1/encoder.js:(anonymous_18)	IF-CC
7	7	lib/v1/encoder.js:(anonymous_19)	CF-CHG
1	7,5	lib/v2/encoder.js:(anonymous_11)	MC-DNP, SQ-AMO
1	7,5	lib/v2/encoder.js:(anonymous_12)	MC-DNP, SQ-RMO
6	8	lib/v1/decoder.js:(anonymous_20)	IF-APC
2	9	lib/v1/decoder.js:(anonymous_20)	IF-CC
1	41,5	lib/v2/encoder.js:Encoder	CF-ADD
5	56,5	lib/v2/encoder.js:(anonymous_4)	IF-APC

For all nine bugs from the `Hessian.js` project, we determined, using the Tarantula score, the ranks for each method affected by the bug-fixes, and we compared these values with the bug-fix patterns we found. Table III lists the bug ID in BUGSJS, the rank obtained from the Tarantula score, the corresponding methods in the bug-fix and the associated bug-fix patterns.

The first method (`lib/v2/encoder.js:(anonymous_3)`) does not have any patterns assigned because the fix only contains an assert statement which is not among the patterns proposed by Pan et al. [15]. We can observe that two methods falling within the IF-CC pattern (`lib/utils.js:(anonymous_3)`, `lib/v1/encoder.js:(anonymous_18)`) have generally better ranks. In these cases, the fix involved adding or modifying existing IF conditions. The only exception is `lib/v1/decoder.js:(anonymous_20)`, where the fix only involved removing a condition. The ranks of fixes falling within the IF-APC(J) pattern are rather different, whereas the occurrence of other patters is not significant. Thus, further experiments with more bug information (for instance, those of the other projects available in BUGSJS) are required to draw conclusions about the correlation between bug patterns and fault localization techniques.

B. Preliminary results: ranking of methods

The previous section presented all bugs from the `Hessian.js` project. Additionally, let us consider Bug-2 of this project[1], which contains 175 tests, 95 methods, 1040 LOC, the method-level coverage is around 98% and the test-method ratio is ≈35:19. These statistics (i.e., non-trivial number of tests and code elements with balanced ratio) enable a more reliable application of our fault localization technique.

The bug-fixing commit changed nine (9) lines of JavaScript code. Based on the location of these changes, we can precisely identify the modified methods (as BUGSJS's coverage data contain information about the starting LOC of

[1]https://github.com/BugsJS/hessian.js/releases/tag/Bug-2-fix

TABLE IV: Hessian.js Bug-2 changes and scores

Method	m_{ef}	m_{ep}	m_{nf}	m_{np}	Tarantula	Rank
anonymous_15	1	6	1	167	**0.93514**	**2**
anonymous_20	1	23	1	150	0.78995	9

each method). In this case, the bug-fixing commit involves two methods: `lib/v1/decoder.js:(anonymous_20)` and `lib/v2/decoder.js:(anonymous_15)`. Table IV reports the four metrics required to compute Tarantula values and the final scores for each of these methods. Results are ranked according to increasing Tarantula scores, hence showing which method is more likely to be buggy. In our example, `anonymous_20` is ranked ninth and `anonymous_15` is ranked second in the order of all methods. Since the bug-fixing commit involves *multiple* methods, the lowest rank associated with all changed methods determines which is the rank of the bug. In this case, it will be two, that is the rank of `anonymous_15`. Since the identified buggy element also included fixes to an if-construct, this is aligned with our observation from the previous section.

C. An additional observation

Unfortunately, we observed that programs can also include failing tests that are not related to the investigated bug. The unrelated tests can interfere with the evaluation of FL algorithms, as such they they must be filtered out. One possible way to do it is by comparing the test results of the buggy and fixed versions: tests that fail in both cases are irrelevant to the examined bug. For example, `Hessian.js` contains six (6) such tests, on average.

V. Conclusions and Future Work

In this abstract paper we have sketched our on-going work on the BugsJS benchmark which has the ultimate purpose of representing a consolidated resource available to researchers in the areas of software analysis and testing for JavaScript. We have also reported an example of possible experiment in fault localization for JavaScript using BugsJS.

We hope our framework can foster further research in this domain, and that can be used to conduct highly-reproducible empirical studies in various areas such as regression testing, bug prediction, and fault localization.

As part of our ongoing and future work, we plan to include more subjects (and corresponding bugs) to the benchmark. Also, we plan to implement easier filtering of such irrelevant test cases, so as to reduce the false positive ration when performing experiments in fault localization. Our long-term goal is to also include client-side JavaScript web applications in BugsJS. Furthermore, we are planning to develop an abstraction layer to allow easier extensibility of our infrastructure to other JavaScript testing frameworks.

References

[1] S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Understanding asynchronous interactions in full-stack JavaScript," in *Proc. of 38th International Conference on Software Engineering (ICSE)*, 2016.

[2] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proc. of 39th International Conference on Software Engineering (ICSE)*, 2017.

[3] 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, 2016.

[4] F. S. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "A Study of Causes and Consequences of Client-Side JavaScript Bugs," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 128–144, Feb 2017.

[5] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Árpád Beszédes, R. Ferenc, and A. Mesbah, "BugJS: A benchmark of javascript bugs," in *Proceedings of 12th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2019. [Online]. Available: https://bugsjs.github.io/paper/ICST19.pdf

[6] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Engg.*, vol. 10, no. 4, pp. 405–435, Oct. 2005.

[7] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. of 2014 International Symposium on Software Testing and Analysis*, 2014.

[8] 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 (TSE)*, vol. 41, no. 12, pp. 1236–1256, December 2015.

[9] N. Dmeiri, D. A. Tomassi, Y. Wang, A. Bhowmick, Y.-C. Liu, P. Devanbu, B. Vasilescu, and C. Rubio-Gonzalez, "BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes," in *Proc. of 41st International Conference on Software Engineering (ICSE)*, 2019.

[10] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge," in *Proc. of International Conference on Systems, Programming, Languages, and Applications: Software for Humanity: Companion*.

[11] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *Proc. of 34th International Conference on Software Engineering (ICSE)*, 2012.

[12] A. Gkortzis, D. Mitropoulos, and D. Spinellis, "VulinOSS: A dataset of security vulnerabilities in open-source systems," in *Proc. of 15th International Conference on Mining Software Repositories*, 2018.

[13] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, Oct 2003, pp. 30–39.

[14] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE*

Transactions on Reliability, vol. 63, no. 1, pp. 290–308, March 2014.

[15] K. Pan, S. Kim, and E. J. Whitehead, “Toward an understanding of bug fix patterns,” *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, Jun 2009.