Supporting JavaScript Experimentation with BugsJS

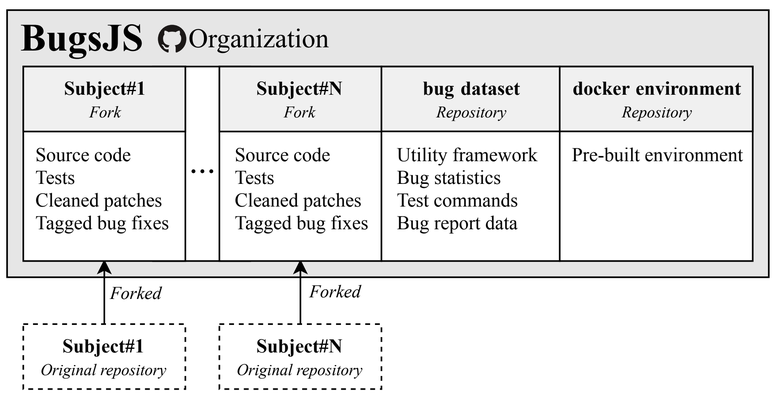
Béla Vancsics, Péter Gyimesi, Andrea Stocco, Davood Mazinanian,
Árpád Beszédes, Rudolf Ferenc, Ali Mesbah

Motivation

JavaScript is the de-facto programming language for the web, and the most adopted one on GitHub. However, JavaScript is also error-prone due to its asynchronous, dynamic, and loosely typed nature.

To support the evaluation of analysis and testing techniques for this langauge, we created BugsJS, a *benchmark of real JavaScript bugs* with corresponding test cases and other artifacts. According to our literature review, BugsJS is the very first benchmark available for the JavaScript domain.

BugsJS



We manually selected and validated *453* JavaScript bugs from *10* JavaScript Node.js programs pertaining to the Mocha testing framework. The bugs are all *manually validated* and come with a comprehensive report and one or more test cases that demonstrate the bug. We have used the GitHub's *fork* feature to include the entire history of the projects in BugsJS. For each bug, we created and tagged three additional commits on top of the buggy version: only the test changes applied; only the production code fix applied; both the cleaned fix and the test changes applied. We also developed a *Docker-based infrastructure* to download, analyze, and run test cases exposing each bug and the corresponding real fixes implemented by developers.

	kLOC	Stars	Commits	Forks	Bugs
Bower	16	15,290	2,706	1,995	3
ESLint	240	12,434	6,615	2,141	333
Express	11	40,407	5,500	7,055	27
Hessian.js	6	104	217	23	9
Hexo	17	23,748	2,545	3,277	12
Karma	12	10,210	2,485	1,531	22
Mongoose	65	17,036	9,770	2,457	29
Node-redis	11	10,349	1,242	1,245	7
Pencilblue	46	1,596	3,675	276	7
Shield	20	6,319	2,036	1,432	4
Total	444	137,493	36,791	21,432	453

Availability

BugsJS is available at
`https://bugsjs.github.io`

Possible Use Cases

Testing techniques: BugsJS includes more than 25k JavaScript test cases, which can be used for different regression testing studies, such as test prioritization, software oracles, or automated test repair.

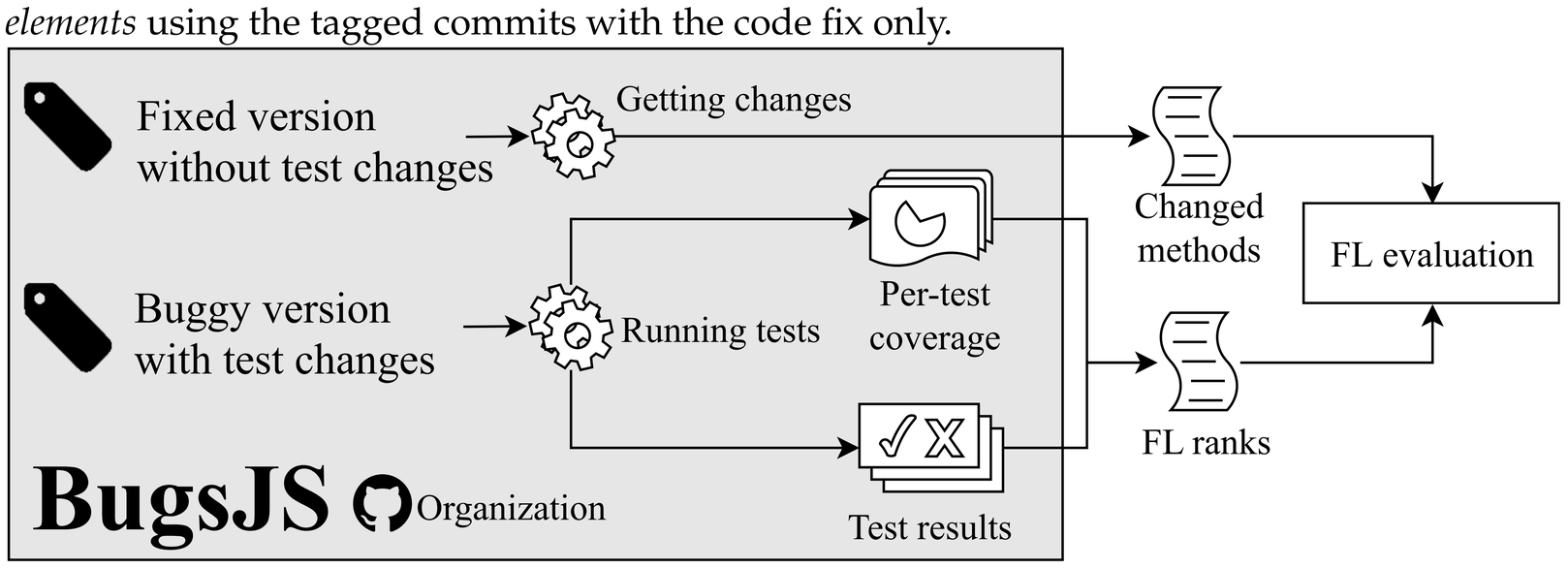
Bug prediction: BugsJS includes source code various information pertaining to a large set of bugs, which can be used to construct bug prediction models. The availability of both uncleaned and cleaned bug-fixing patches in the dataset can allow assessing the sensitivity of the proposed models to noise.

Bug localization: BugsJS contains pointers to the natural language bug description and discussions for several hundreds of bugs. Text retrieval techniques such as NLP can be also used to formulate a natural language query that describes the observed bug, since bugs are readily available. Also, Spectrum Based Fault Localization experiments can be easily performed.

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.

Example Application and Preliminary Evaluation

We applied a Spectrum-Based Fault Localization (SBFL) technique, namely Tarantula, to one of the subject programs in BugsJS, `Hessian.js`, which includes nine bugs. BugsJS's API provides an easy access to all information required for such an experiment: *per-test coverage* using the `per-test` command, *test results* using the `test` command, and *modified source code elements* using the tagged commits with the code fix only.



We categorized the 9 bugs of the `Hessian.js` project based on recurring bug-fix patterns and compared it with the results of the SBFL technique, hoping to gather insights on how different types of bugs can be successfully localized. The Table below lists the Tarantula ranks next the associated bug-fix patterns. The results suggest that methods with the IF-CC pattern have generally better ranks, however further experiments with more bug information are required to draw conclusions about the correlation between bug patterns and fault localization techniques.

Bug #	Rank	Method	Pattern(s)
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

Let us consider Bug-2 of this project. Based on the bug-fixing commit that changed 9 lines of code, we can precisely identify the modified methods. In this case, it involves two methods: `lib/v1/decoder.js:(anonymous_20)` and `lib/v2/decoder.js:(anonymous_15)`. The Table below 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. 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 the rank of the bug which in this case is two.

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

Future Enhancements

Expansion: We would like to add new, server-side JavaScript programs to the benchmark, thus increasing the number of existing bugs.

Extension: In addition, we want to expand the scope of the programs examined by client-side JavaScript web applications.

Support: We are planning to develop the framework, especially with the support of other testing frameworks, so we can add new programs to the examination (and to the dataset).