BugsJS: A Benchmark of JavaScript Bugs

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

*University of Szeged, Hungary
{pgyimesi, vancsics, beszedes, ferenc}@inf.u-szeged.hu

†University of British Columbia, Canada
{astocco, dmazinanian, amesbah}@ece.ubc.ca

Abstract—**JavaScript is a popular programming language that is also error-prone due to its asynchronous, dynamic, and loosely-typed nature. In recent years, numerous techniques have been proposed for analyzing and testing JavaScript applications. However, our survey of the literature in this area revealed that the proposed techniques are often evaluated on different datasets of programs and bugs. The lack of a commonly used benchmark limits the ability to perform fair and unbiased comparisons for assessing the efficacy of new techniques. To fill this gap, we propose BugsJS, a benchmark of 453 real, manually validated JavaScript bugs from 10 popular JavaScript server-side programs, comprising 444k LOC in total. Each bug is accompanied by its bug report, the test cases that detect it, as well as the patch that fixes it. BugsJS features a rich interface for accessing the faulty and fixed versions of the programs and executing the corresponding test cases, which facilitates conducting highly-reproducible empirical studies and comparisons of JavaScript analysis and testing tools.**

I. Introduction

JavaScript (JS) is the de-facto web programming language globally[1], and the most adopted language on GitHub[2]. JavaScript is massively used in the client-side of web applications to achieve high responsiveness and user friendliness. In recent years, due to its flexibility and effectiveness, it has been increasingly adopted also for server-side development, leading to full-stack web applications [1]. Platforms such as Node.js[3] allow developers to conveniently develop both the front- and back-end of the applications entirely in JavaScript.

Despite its popularity, the intrinsic characteristics of JavaScript—such as weak typing, prototypal inheritance, and run-time evaluation—make it one of the most error-prone programming languages. As such, a large body of software engineering research has focused on the analysis and testing of JavaScript web applications [2, 3, 4, 5, 6, 7, 8, 9].

Existing research techniques are typically evaluated through empirical methods (*e.g.*, controlled experiments), which need software-related artifacts, such as source code, test suites, and descriptive bug reports. To date, however, most of the empirical works and tools for JavaScript have been evaluated on different datasets of subjects. Additionally, subject programs or accompanying experimental data are rarely made available in a detailed, descriptive, curated, and coherent manner. This not only hampers the reproducibility of the studies themselves, but also makes it difficult for researchers to assess the state-of-the-art of related research and to compare existing solutions.

Specifically, testing techniques are typically evaluated with respect to their effectiveness at detecting faults in existing programs. However, real bugs are hard to isolate, reproduce and characterize. Therefore, the common practice relies on manually-seeded faults, or mutation testing [10]. Each of these solutions has limitations. Manually-injected faults can be biased toward researchers' expectations, undermining the representativeness of the studies that use them. Mutation techniques, on the other hand, allow generating a large number of "artificial" faults. Although research has shown that mutants are quite representative of real bugs [11, 12, 13], mutation testing is computationally expensive to use in practice. For these reasons, a benchmark of manually validated bugs can be of paramount importance for devising novel debugging, fault localization, or program repair approaches.

Several benchmarks of bugs have been proposed and largely utilized by researchers to advance testing research. Notable instances are the Software-artifact Infrastructure Repository [14], Defects4J [15], ManyBugs [16], and BugSwarm [17]. Purpose-specific test and bug datasets also exist to support studies in program repair [18], test generation [19], and security [20]. However, to date, a well-organized repository of labeled JavaScript bugs is still missing. The plethora of different JavaScript implementations available (*e.g.*, V8, JavaScriptCore, Rhino) further makes devising a cohesive bugs benchmark nontrivial.

In this paper, we present BugsJS, a benchmark of 453 JavaScript-related bugs from 10 open-source JavaScript projects, based on Node.js and the Mocha testing framework. BugsJS features an infrastructure containing detailed reports about each bug, the faulty versions of programs, the test cases exposing them, as well as the patches that fix them.

Particularly, this paper makes the following contributions:

- A survey of the previous work on analysis and testing of JavaScript applications, revealing the lack of a comprehensive benchmark of JavaScript programs and bugs to support empirical evaluation of the proposed techniques.
- BugsJS, a benchmark of 453 manually selected and validated JavaScript bugs from 10 JavaScript Node.js

[1]https://insights.stackoverflow.com/survey/2018
[2]https://octoverse.github.com
[3]https://nodejs.org/en/

programs pertaining to the Mocha testing framework. BugsJS features a Docker-based infrastructure to download, analyze, and run test cases exposing each bug and the corresponding real fixes implemented by developers.
- A quantitative and qualitative analysis of the bugs included in BugsJS, illustrating their generalizability to existing taxonomies of faults and fixes.

II. Studies on JavaScript analysis and testing

To motivate the need for a novel benchmark for JavaScript bugs, we surveyed the works related to software analysis and testing in the JavaScript domain. Our review of the literature also allowed us to gain insights about the most active research areas in which our benchmark should aim to be useful.

In the JavaScript domain, the term benchmark commonly refers to collections of programs used to measure and test the performance of web browsers with respect to the latest JavaScript features and engines. Instances of such performance benchmarks are JetStream,[4] Kraken,[5] Dromaeo,[6] Octane,[7] and V8.[8] In this work, however, we refer to *benchmark* as a collection of JavaScript programs and artifacts (*e.g.*, test cases or bug reports) used to support empirical studies (*e.g.*, controlled experiments or user studies) related to one or more research areas in software analysis and testing.

We used the databases of scientific academic publishers and popular search engines to look for papers related to different software analysis and testing topics for JavaScript. We adopted various combinations of keywords: `JavaScript`, `testing` (including code coverage measurement, mutation testing, test generation, unit testing, test automation, regression testing), `bugs` and `debugging` (including fault localization, bug and error classification), and `web`. We also performed a lightweight forward and backward snowballing to mitigate the risk of omitting relevant literature.

Last, we examined the evaluation section of each paper. We retained only papers in which real-world, open-source JavaScript projects were used, whose repositories and versions could be clearly identified. This yielded 25 final papers. Nine (9) of these studies are related to bugs, in which 670 subjects were used in total. The remaining 16 papers are related to other testing fields, comprising 494 subjects in total.

In presenting the results of our survey of the literature, we distinguish (1) studies containing specific bug information and other artifacts (such as source code and test cases) and (2) studies containing only JavaScript programs and other artifacts not necessarily related to bugs.

A. Bug-related Studies for JavaScript

We analyzed papers using JavaScript systems that include bug data in greater detail, because these works can provide us important insights about the kind of analysis researchers used the subjects for, and thus, the requirements that a new benchmark of bugs should adhere to.

We found nine studies in this category, which we overview next. Ocariza et al. [5] present an analysis and classification of bug reports to understand the root causes of client-side JavaScript faults. This study includes 502 bugs from 19 projects with over 2M LOC. Another bug classification presented by Gao et al. [21] focuses on type system-related issues in JavaScript, which includes about 400 bug reports from 398 projects with over 7M LOC. Hanam et al. [22] present a study of cross-project bug patterns in server-side JavaScript code, using 134 Node.js projects of about 2.5M LOC.

Ocariza et al. [23] propose an inconsistency detection technique for MVC-based JavaScript applications which is evaluated on 18 bugs from 12 web applications (7k LOC). A related work [24] uses 15 bugs in 20 applications (nearly 1M LOC). They also present an automated technique to localize JavaScript faults based on a combination of dynamic analysis, tracing, and backward slicing, which is evaluated on 20 bugs from 15 projects (14k LOC) [25]. Also, their technique for suggesting repairs for DOM-based JavaScript faults is evaluated on 22 bugs from 11 applications (1M LOC) [26].

Wang et al. [4] present a study on 57 concurrency bugs in 53 Node.js applications (about 3.5M LOC). The paper proposes several different analyses pertaining to the retrieved bugs, such as bug patterns, root causes, and repair strategies. Davis et al. [27] propose a fuzzing technique for identifying concurrency bugs in server-side event-driven programs, and evaluate their technique on 12 real world programs (around 216k LOC) and 12 manually selected bugs.

B. Other Analysis and Testing Studies for JavaScript

Empirical studies in software analysis and testing benefit from a large variety of software artifacts other than bugs, such as test cases, documentation, or code revision history. In this section, we briefly describe the remaining papers of our survey.

Milani Fard and Mesbah [28] *characterize JavaScript tests* in 373 JavaScript projects according to various metrics, *e.g.*, code coverage, test commits ratio, and number of assertions.

Mirshokraie et al. propose several approaches to JavaScript automated testing. This includes an *automated regression testing* based on dynamic analysis, which is evaluated on nine web applications [29]. The authors also propose a *mutation testing* approach, which is evaluated on seven subjects [30], and on eight applications in a related work [31]. They also propose a technique to aid *test generation* based on program slicing [32], where unit-level assertions are automatically generated for testing JavaScript functions. Seven open-source JavaScript applications are used to evaluate their technique. The authors also present a related approach for JavaScript unit test case generation, which is evaluated on 13 applications [33].

Adamsen et al. [34] present a hybrid static/dynamic program analysis method to check *code coverage-based properties* of test suites from 27 programs. Dynamic symbolic execution is used by Milani Fard et al. [35] to generate DOM-based *test fixtures and inputs* for unit testing JavaScript functions,

[4]https://browserbench.org/JetStream
[5]https://wiki.mozilla.org/Kraken
[6]https://wiki.mozilla.org/Dromaeo
[7]https://developers.google.com/octane
[8]https://github.com/hakobera/node-v8-benchmark-suite

TABLE I: Subject distribution among surveyed papers

Bug-related		All studies	
# Papers	# Subjects	# Papers	# Subjects
1	607	1	910
2	17	2	91
3	7	3	17
4	2	4	4
5	0	5	1

and four experimental subjects are used for evaluation. Ermuth and Pradel propose a GUI test generation approach [6], and evaluate it on four programs.

Artzi et al. [36] present a framework for *feedback-directed automated test generation* for JavaScript web applications. In their study, the authors use 10 subjects. Mesbah et al. [37] present Atusa, a *test generation technique for Ajax-based applications* which they evaluate on six web applications. A comprehensive survey of *dynamic analysis and test generation* for JavaScript is presented by Andreasen et al. [38].

Billes et al. [2] present a black-box analysis technique for multi-client web applications to detect *concurrency errors* on three real-world web applications. Hong et al. [39] present a testing framework to detect concurrency errors in client-side web applications written in JavaScript, and use five real-world web applications.

Wang et al. [3] propose a modification to the *delta debugging* approach that reduces the event trace, which is evaluated on 10 real-world JavaScript application failures. Dhok et al. [40] present a *concolic testing* approach for JavaScript which is evaluated on 10 subjects.

C. Findings

In the surveyed papers, we observed that the proposed techniques were evaluated using different sets of programs, with little to no overlap.

Table I shows the program distribution per paper. In bug-related studies, 670 subject programs were adopted overall, of which 633 were unique. 607 of these programs (96%) were used in only one study, and no subject was used in more than four papers (Table I, columns 1 and 2). Other studies exhibit the same trend (Table I, columns 3 and 4): overall, 1,164 subjects were used in all the investigated papers, of which 1,023 were unique. From these, 910 (89%) were used in only one paper, and no subject was used in more than five papers.

In conclusion, we observe that the investigated studies involve different sets of programs, since no centralized benchmark is available to support reproducible experiments in analysis and testing related to JavaScript bugs. To fill this gap, in this paper we propose BugsJS, a benchmark of real JavaScript bugs, whose design and implementation is described next.

III. BugsJS – the Proposed Benchmark

To construct a benchmark of real JavaScript bugs, we identify existing bugs from the programs' version control histories, and collect the real fixes provided by developers. Developers often manually label the revisions of the programs in which reported bugs are fixed (*bug-fixing commits*, or patches). As such, we refer to the revision preceding the bug-fixing commit as the *buggy commit*. For the purpose of the benchmark, this allowed us to extract detailed bug reports and descriptions, along with the buggy and bug-fixing commits they refer to.

Each bug and fix should adhere to the following properties:

- **Reproducibility.** One or more *test cases* are available in a *buggy commit* to demonstrate the bug. The bug must be reproducible under reasonable constraints. For this reason, we excluded non-deterministic features and flaky tests from our study, since replicating them in a controlled environment would be excessively challenging.
- **Isolation.** The bug-fixing commit applies to JavaScript source code files only; changes to other artifacts such as documentation or configuration files are not considered. The source code of each commit must be *cleaned* from irrelevant changes (*e.g.*, feature implementations, refactorings, changes to non-JavaScript files). The *isolation* property is particularly important in research areas where the presence of noise in the data has detrimental impacts on the techniques (*e.g.*, automated program repair, or fault localization approaches).

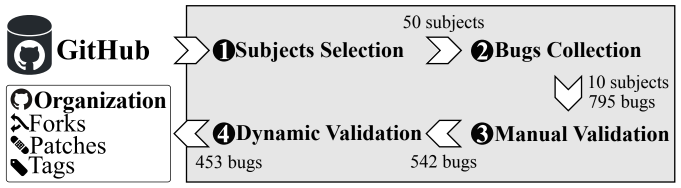


Fig. 1: Overview of the bug selection and inclusion process.

Figure 1 depicts the main steps of the process we performed to construct our benchmark. First, we adopted a systematic procedure to select the JavaScript subjects to extract the bug information from. Then, we collected bug candidates from the selected projects, and manually validated each bug for inclusion by means of multiple criteria. Next, we performed a dynamic sanity check to make sure that the tests introduced in a bug-fixing commit can detect the bug in the absence of its fix. Finally, the retained bugs were cleaned from irrelevant patches (*e.g.*, comments, or whitespaces).

A. Subject Systems Selection

To select appropriate subject systems to include in BugsJS, we focused on popular and trending JavaScript projects on GitHub. Such projects often engage large communities of developers, and therefore, are more likely to follow best software development practices, including bug reporting and tracking. Moreover, GitHub's *issue IDs* allow conveniently connecting bug reports to bug-fixing commits.

Popularity was measured using the projects' *Stargazers count* (*i.e.*, the number of stars owned by the subject's GitHub repository). We selected server-side Node.js applications which are popular (Stargazers count $\geq$ 100) and mature (number of commits $>$ 200), and have been actively

TABLE II: Subjects included in BUGSJS

PROGRAM		STATS				TESTS (#)				COVERAGE (%)			
Name	**Description**	**kLOC (JS)**	**Stars**	**Commits**	**Forks**	**All**	**Passing**	**Pending**	**Failing**	**Statements**	**Branches**	**Functions**	**Lines**
BOWER[9]	package manager	16	15,290	2,706	1,995	455	103	19	36	81.11	66.91	80.62	81.11
ESLINT[10]	linting tool	240	12,434	6,615	2,141	18,528	18,474	0	54	99.21	98.19	99.72	99.21
EXPRESS[11]	web framework	11	40,407	5,500	7,055	855	855	0	0	98.71	94.32	100	99.95
HESSIAN.JS[12]	serialization service	6	104	217	23	225	223	2	0	96.42	91.27	98.99	96.42
HEXO[13]	blog framework	17	23,748	2,545	3,277	875	868	7	0	96.20	90.51	98.54	97.27
KARMA[14]	test runner	12	10,210	2,485	1,531	331	331	0	0	54.61	34.03	43.98	54.76
MONGOOSE[15]	ODM	65	17,036	9,770	2,457	2,107	2,071	36	0	90.97	85.95	89.65	91.04
NODE-REDIS[16]	database client	11	10,349	1,242	1,245	966	965	0	1	99.06	98.19	97.99	99.06
PENCILBLUE[17]	CMS	46	1,596	3,675	276	807	802	0	5	35.21	19.09	22.91	35.22
SHIELDS[18]	badge service	20	6,319	2,036	1,432	482	469	13	0	75.98	65.60	83.26	75.97

maintained (year of the latest commit $\geq$ 2017). We currently focus on Node.js because it is emerging as one of the most pervasive technologies to enable using JavaScript in the server side, leading to the so-called full-stack web applications [1].

We examined the GitHub repository of each retrieved subject system to ensure that bugs were properly tracked and labeled. Particularly, we only selected projects in which bug reports had a dedicated *issue label* on GitHub's *Issues* page, which allows filtering irrelevant issues (pertaining to, *e.g.*, feature requests, build problems, or documentation tasks), so that only *actual bugs* are included.

Our initial list of subjects included 50 Node.js programs, from which we filtered out projects based on the number of candidate bugs found and the adopted testing frameworks.

B. Bugs Collection

Collecting bugs and bug-fixing commits. For each subject system, we first queried GitHub for *closed issues* assigned with a specific bug label using the official GitHub's API.[19] For each closed bug, we exploit the *links* existing between issues and commits to identify the corresponding bug-fixing commit. GitHub automatically detects these links when there is a specific keyword (belonging to a predefined list[20]), followed by an issue ID (*e.g.*, `Fixes #14`).

Each issue can be linked to zero, one, or more source code commits. A closed bug without a bug-fixing commit could mean that the bug was rejected (*e.g.*, it could not be replicated), or that developers did not associate that issue with any commit. We discarded such bugs from our benchmark, as we require each bug to be identifiable by its bug-fixing commit. At last, similarly to existing benchmarks [15], we discarded bugs linked to more than one bug-fixing commit, as this might imply that they were fixed in multiple steps, or that the first attempt for fixing them was unsuccessful.

Including corresponding tests. We require each fixed bug to have *unit tests* that demonstrate the absence of the bug. To meet this requirement, we examined the bug-fixing patches to ensure they also contain changes or additions in the test files. For this filtering, we manually examined each patch to determine whether test files were involved. The result of this step is the list of *bug candidates* for the benchmark. From the initial list of 50 subject systems, we considered the projects having at least 10 bug candidates.

Testing frameworks. There are several testing frameworks available for JavaScript applications. We collected statistics about the testing frameworks used by the 50 considered JavaScript projects. Our results show that there is no single predominant testing framework for JavaScript (as compared to, for instance, JUnit which is used by most Java developers). We found that the majority of tests in our pool were developed using Mocha[21] (52%), Jasmine[22] (10%), and QUnit[23] (8%). The prevalence of Mocha as the most popular JavaScript testing framework was also supported in a previous large-scale empirical study [28]. Consequently, the initial version of BUGSJS only includes projects that use Mocha, and extending to other JavaScript testing frameworks (*e.g.*, for client-side testing) is left for future work.

Final Selection. Table II reports the names and descriptive statistics of the 10 applications we ultimately retained. Notice that all these applications have at least 1000 LOC (frameworks excluded), thus being representative of modern web applications (Ocariza et al. [5] report an average of 1,689 LOC for AngularJS web applications on GitHub with at least 50 stars).

C. Manual Patch Validation

We manually investigated each bug and the corresponding bug-fixing commit to ensure that only bugs meeting certain criteria are included, as described below.

[9]https://github.com/bower/bower
[10]https://github.com/eslint/eslint
[11]https://github.com/expressjs/express
[12]https://github.com/node-modules/hessian.js
[13]https://github.com/hexojs/hexo
[14]https://github.com/karma-runner/karma
[15]https://github.com/Automattic/mongoose
[16]https://github.com/NodeRedis/node_redis
[17]https://github.com/pencilblue/pencilblue
[18]https://github.com/badges/shields
[19]https://developer.github.com/v3/
[20]https://help.github.com/articles/closing-issues-using-keywords/
[21]https://mochajs.org/
[22]https://jasmine.github.io/
[23]https://qunitjs.com/

TABLE III: Bug-fixing commit inclusion criteria

Rule Name	Description
Isolation	The bug-fixing changes must fix only one (1) bug (i.e., must close exactly one (1) issue)
Complexity	The bug-fixing changes should involve a limited number of files ($\leq$ 3), lines of code ($\leq$ 50) and be understandable within a reasonable amount of time (max 5 minutes)
Dependency	If a fix involves introducing a new dependency (e.g., a library), there must also exist production code changes and new test cases added in the same commit
Relevant Changes	The bug-fixing changes must involve only changes in the production code that aim at fixing the bug (whitespace and comments are allowed)
Refactoring	The bug-fixing changes must not involve refactoring of the production code

Methodology. Two authors of this paper manually investigated each bug and its corresponding bug-fixing commit and labeled them according to a well-defined set of *inclusion criteria* (Table III). The bugs that met all the criteria were initially marked as "Candidate Bug" to be considered for inclusion in the benchmark.

In detail, for each bug, the authors investigated simultaneously the code of the commit to ensure relatedness to the bug being fixed. During the investigation, however, several bug-fixing commits were too complex to comprehend by the investigators, either because domain knowledge was required, or because the number of files or lines of code being modified was large. We labeled such complex bug-fixing commits as "Too complex", and discarded them from the current version of BugsJS. The rationale is to keep the size of the patches within reasonable thresholds, so as to select a high quality corpus of bugs which can be easily analyzable and processable by both manual inspection and automated techniques. Particularly, we deemed a commit being too complex if the production code changes involved more than three (3) files or more than 50 LOC, or if the fix required more than 5 minutes to understand. In all these cases, a discussion was triggered among the authors, and the case was ignored only if the authors unanimously decided that the fix was too complex.

Results. Overall, we manually validated 795 commits (*i.e.*, bug candidates), of which 542 (68.18 %) fulfilled the criteria. Table IV (Manual) illustrates the result of this step for each application and across all applications.

The most common reason for excluding a bug is that the fix was deemed as too complex (136). Other frequent scenarios include cases where a bug-fixing commit addressed more than one bug (32), or where the fix did not involve production code (29), or contained refactoring operations (39). Also, we found four cases in which the patch did not involve the actual test's source code, but rather comments or configuration files.

D. Sanity Checking through Dynamic Validation

To ensure that the test cases introduced in a bug-fixing commit were actually intended to test the buggy feature, we adopted a systematic and automatic approach described next.

Methodology. Let V_{bug} be the version of the source code that contains a bug b, and let V_{fix} be the version in which b is fixed. The existing test cases in V_{bug} do not fail due to b. However, at least one test of V_{fix} should fail when executed on V_{bug}. This allows us to identify the test in V_{fix} used to demonstrate b (*isolation*) and to discard cases in which tests immaterial to the considered buggy feature were introduced.

TABLE IV: Manual and dynamic validation statistics per application for all considered commits

		Bower	Eslint	Express	Hessian.js	Hexo	Karma	Mongoose	Node-redis	Pencilblue	Shields	**Total**
	Initial number of bugs	10	559	39	17	24	37	56	25	18	10	**795**
Manual	✘ Fixes multiple issues	0	18	1	0	1	5	2	5	0	0	32
	✘ Too complex	0	94	0	4	8	4	8	7	9	2	136
	✘ Only dependency	1	9	0	0	1	0	2	0	0	0	13
	✘ No production code	0	20	4	0	1	1	2	0	0	1	29
	✘ No tests changed	1	0	1	0	0	0	0	1	1	0	4
	✘ Refactoring	0	36	0	0	0	1	1	1	0	0	39
	After manual validation	8	382	33	13	13	26	41	11	8	7	**542**
Dynamic	✘ Test does not fail at V_{bug}	1	11	6	4	1	2	8	3	1	3	40
	✘ Dependency missing	3	17	0	0	0	1	1	0	0	0	22
	✘ Error in tests	1	7	0	0	0	0	3	1	0	0	12
	✘ Not Mocha	0	14	0	0	0	1	0	0	0	0	15
	✔ ***Final Number Of Bugs***	3	333	27	9	12	22	29	7	7	4	**453**

To run the tests, we obtained the dependencies and set up the environment for each specific revision of the source code. Over time, however, developers made major changes to some of the projects' structure and environment, making tests replication infeasible. These cases occurred, for instance, when older versions of required dependencies were no longer available, or when developers migrated to a different testing framework (*e.g.*, from QUnit to Mocha).

For the projects that used scripts (*e.g.*, `grunt`, `bash`, `Makefile`) to run their tests, we extracted them, so as to isolate each test's execution and avoiding possible undesirable side effects caused by running the complete test suite.

Results. After the dynamic analysis, 453 bug candidates were ultimately retained for inclusion in BugsJS (84% of the 542 bug candidates from the previous step).

Table IV (Dynamic) reports the results for the dynamic validation phase. In 22 cases, we were unable to run the tests because dependencies were removed from the repositories. In 15 cases, the project at revision V_{bug} did not use Mocha for testing b. In 12 cases, tests were failing during the execution, whereas in 40 cases no tests failed when executed on V_{bug}. We excluded all such bug candidates from the benchmark.

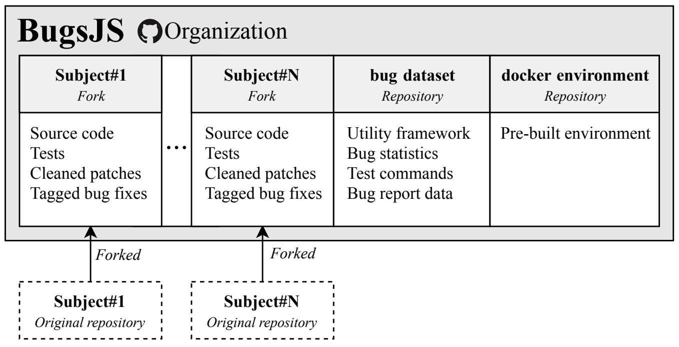


Fig. 2: Overview of BugsJS architecture

E. Patch Creation

We performed *manual cleaning* on the bug-fixing patches, to make sure they only include changes related to bug fixes. In particular, we removed *irrelevant files* (*e.g.*, `*.md`, `.gitignore`, `LICENSE`), and *irrelevant changes* (*i.e.*, source code comments, when only comments changed, and comments unrelated to bug-fixing code changes, as well as changes solely pertaining to whitespaces, tabs, or newlines). Furthermore, for easier analysis, we separated the patches into two separate files, the first one including the modifications to the tests, and the second one pertaining to the production code fix.

F. Final Benchmark Infrastructure and Implementation

Infrastructure. Figure 2 illustrates the overall architecture of BugsJS, which supports common activities related to the benchmark, such as running the tests at each revision, or checking out specific commits. The framework's command-line interface includes the following commands:

- `info`: Prints out information about a given bug.
- `checkout`: Checks-out the source code for a given bug.
- `test`: Runs all tests for a given bug and measures the test coverage.
- `per-test`: Runs each test individually and measures the per-test coverage for a given bug.

For the `checkout`, `test`, and `per-test` commands, the user can specify the desired code revision: *buggy*, *buggy with the test modifications applied*, or the *fixed version*.

BugsJS is equipped with a pre-built environment that includes the necessary configurations for each project to execute correctly. This environment is available as a Docker image along with a detailed step-by-step tutorial.

The interested reader can find BugsJS, all subject systems and related bugs in the dedicated GitHub organization:

https://github.com/BugsJS/

Source code commits and tests. We used GitHub's *fork* functionality to make a full copy of the *git* history of the subject systems. The unique identifier of each commit (*i.e.*, the commit `SHA1` hashes) remains intact when forking. In this way, we were able to synchronize the copied fork with the original repository and keep it up-to-date. Importantly, our benchmark will not be lost if the original repositories get deleted.

The fork is a separate *git* repository; therefore, we can push commits to it. Taking advantage of this possibility, we have extended the repositories with additional commits, to separate the bug-fixing commits and their corresponding tests. To make such commits easily identifiable, we tagged them using the following notation (`X` denotes a sequential bug identifier):

- `Bug-X`: The parent commit of the revision in which the bug was fixed (*i.e.*, the buggy revision);
- `Bug-X-original`: A revision with the original bug-fixing changes (including the production code and the newly added tests);
- `Bug-X-test`: A revision containing only the tests introduced in the bug-fixing commit, applied to the buggy revision;
- `Bug-X-fix`: A revision containing only the production code changes introduced to fix the bug, applied to the buggy revision;
- `Bug-X-full`: A revision containing both the cleaned fix and the newly added tests, applied to the buggy revision.

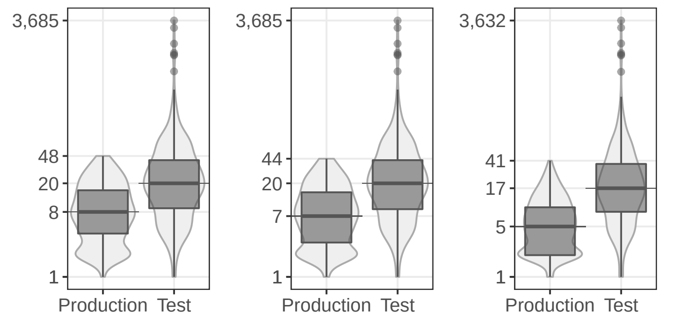
Test runner commands. For each project, we have included the necessary test runner commands in a CSV file. Each row of the file corresponds to a bug in the benchmark, and contains the following information:

1) A sequential bug identifier;
2) The test runner command required to run the tests;
3) The test runner command required to produce the test coverage results;
4) The Node.js version required for the project at the specific revision where the bug was fixed, so that the tests can execute properly;
5) The preparatory test runner commands (*e.g.*, to initialize the environment to run the tests, which we call `pre-commands`);
6) The cleaning test runner commands (*e.g.*, the tear down commands, which we call `post-commands`) to restore the application's state.

Test coverage data. Furthermore, for each project, we included in BugsJS some pre-calculated information about the tests of the `Bug-X` versions in a separate CSV file. Each row of the file contains the following information:

1) A sequential bug identifier;
2) Total LOC in the source code, as well as LOC covered by the tests;
3) The number of functions in the source code, as well as the number of functions covered by the tests;
4) The number of branches in the source code, as well as the number of branches covered by the tests;
5) The total number of tests in the test suite, along with the the number of passing, failing and pending tests (*i.e.*, the tests which were skipped due to execution problems).

Bug report data. Forking repositories does not maintain the issue data associated with the original repository. Thus,



(a) Before cleaning (b) Cleaned (c) Cleaned+No Comments

Fig. 3: Distribution of the churn in bug-fixing commits.

the links appearing in the commit messages of the forked repository still refer to the original issues. In order to preserve the bug reports, we obtained them via the GitHub's API and stored them in the Google's Protocol Buffers[24] format. Particularly, for each bug report, we store the original issue identifier paired with our sequential bug identifier, the text of the bug description, the issue open and close dates, the SHA1 of the original bug-fixing commit along with the commit date and commit author identifiers. Lastly, we save the comments from the issues' discussions.

IV. Analysis

To gain a better understanding about the characteristics of the bugs and fixes included in BugsJS, we have performed two analyses to quantitatively and qualitatively assess the representativeness of our benchmark.

A. Code churn

Code *churn* is a measure that approximates the rate at which code evolves. It is defined as the sum of the number of lines added and removed in a source code change. The churn is an important measure with several uses in software engineering studies, *e.g.*, as a direct or indirect predictor in bug prediction models [41, 42].

We utilize the distribution of code churn to describe the overall data distribution in the benchmark, and understand to what extent it can be used to support software testing techniques (*e.g.*, fault localization, program repair) that are directly affected by the size of the source code changes.

Figure 3 illustrates the distribution of code churn in each bug-fixing commit in BugsJS, *before* cleaning the patches (Figure 3a), *after* removing changes unrelated to the fix, while retaining related comments added or removed during the bug fix (Figure 3b), and *after cleaning* the fixes and also removing all the comments and whitespaces (Figure 3c). The box and whisker plots show the quartiles of the data, enhanced by the underlying violin plots to better depict the data distribution.

The median value for the code churn *required to fix a bug* is 5 and 17 for production and test code, respectively (excluding irrelevant changes, comments, and whitespaces). This essentially means that for each line of bug-fixing change in the production code, more than three lines of test code were changed on average.

[24]https://developers.google.com/protocol-buffers/

We can also observe that nearly half of the changes done to the production code involve more than five lines of code, suggesting that existing fault localization and repair techniques will fall short in being applicable on real JavaScript bugs, as they currently deal with one-liner changes only. Previous work showed that the same conclusion holds for Java projects [43].

In addition, comparing the median values in Figure 3 suggests that developers, on average, add/remove one line of unrelated code changes (*i.e.*, $8 - 7$) and two lines of comments/whitespace (*i.e.*, $7 - 5$) when fixing a bug. This essentially shows the importance of the manual isolation and cleaning performed on the bugs included in BugsJS.

The largest value for churn in test code is 3,632 lines (Figure 3c), occurring in a bug-fixing commit in the ESLINT project. This commit corresponds to generated test data committed along with the production code bug-fixing changes. A closer look at the data revealed that there are five other such commits in this project, all changing more than 1,000 lines of test code. Such commits in which the number of changes is exceptionally high (*i.e.*, outliers) should be carefully handled or discarded when conducting empirical studies.

B. Patterns in Bugs and Fixes

We further analyzed the bugs in BugsJS to observe occurrence of repeated *bug patterns*. Recurring patterns can, for example, indicate which categories of bugs merit the greatest attention, and which novel automated repair techniques tailored for JavaScript researchers should focus on. However, in our opinion, a wide range of bugs should be represented, because *diversity* can make the benchmark more suitable to be utilized for different analysis, testing, and repair techniques.

In addition, we carefully looked at low-level *bug fixes* for recurring patterns. Previous work [43, 44, 45] have studied patterns in bug-fixing changes within Java programs. They suggest that the existence of patterns in fixes reveals that specific kinds of code constructs (*e.g.*, if conditionals) could signal weak points in the source code where developers are consistently more prone to introducing bugs [44].

Methodology. Four authors of this paper manually investigated all the 453 bug-fixing commits in BugsJS and attempted to assign the bugs and bug-fixing changes with one of the predefined categories suggested in previous studies. Particularly, we utilized the bug pattern categories proposed by Hanam et al. [22] which is, to our knowledge, the only work proposing a catalogue of bug patterns in server-side JavaScript programs. Our goal was to assess to which extent these patterns apply to the bugs included in BugsJS.

Concerning patterns in bug fixes, on the other hand, we used the categories proposed by Pan et al. [44]. These categories, however, are related to Java bug fixes. Our aim is to assess whether they generalize to JavaScript, or whether, in contrast, JavaScript-specific bug fix patterns would emerge.

TABLE V: Bug pattern categories (Hanam et al. [22])

Bug Pattern	Description	#
Dereferenced non-values	Uninitialized variables	35
Incorrect API config	Missing API call configuration values	3
Incorrect comparison	=== and == used interchangeably	2
Unhanded exceptions	Missing `try-catch` block	2
Missing arguments	Function call with missing arguments	0
Incorrect `this` bounding	Accessing a wrong `this` reference	0

TABLE VI: Bug-fixing change types (Pan et al. [44])

	Category	Example	#
EXISTING	`if`-related	Changing `if` conditions	291
	Assignments	Modifying the RHS of an assignment	166
	Function calls	Adding or modifying an argument	151
	Class fields	Adding/removing class fields	151
	Function declarations	Modifying a function's signature	94
	Sequences	Adding a function call to a sequence of calls, all with the same receiver	42
	Loops	Changing a loop's predicate	7
	`switch` blocks	Adding/removing a switch branch	6
	`try` blocks	Introducing a new `try-catch` block	1
NEW	`return` statements	Changing a `return` statement	40
	Variable declaration	Declaring an existing variable	2
	Initialization	Initializing a variable with empty object literal/array	3

Following the original category definitions [22, 44], we assigned each *individual* bug/fix to exactly one category. Disagreements concerning classification or potential new categories were resolved by further discussion between the authors. To identify the occurrences of such patterns, we opted for a manual analysis to ensure covering potential new patterns, and to add an extra layer of validation against potential misclassifications (*e.g.*, false positives).

1) Bug Patterns: Hanam et al. [22] discuss 13 cross-project bug patterns occurring in JavaScript pertaining to six categories, which are listed in Table V.

Across all 453 bugs of BUGSJS, our analysis found 42 occurrences of the categories proposed by Hanam et al. [22]. The last column of Table V shows the detailed occurrences for each category. Our analysis showed that the majority of the bugs are indeed logical errors made by developers during the implementation which do not necessarily fall into recurring patterns. This essentially shows that the bugs included in BUGSJS are rather diverse in nature, making it ideal for evaluating a wide range of analysis and testing techniques.

In the found patterns, the *Dereferenced non-values* is by far the most prevalent bug pattern (Table V). Previous work showed that this pattern occurs frequently also in client-side JavaScript applications [5]. Developers could avoid these syntax-related bugs by adopting appropriate coding standards. Moreover, IDEs can be enhanced to alert programmers to possible effects or bad practices. They could also aid in prevention by prohibiting certain actions or by recommending the creation of stable constructs.

2) Bug fix patterns: Table VI shows the number of bug fix occurrences followed the categories by Pan et al. [44]. (For fixes spanning multiple lines, we possibly assigned more than one category to a single bug-fixing commit, hence, the overall number of occurrences is greater than the number of bugs.)

Note that, since the categories proposed by Pan et al. have been derived from Java programs, we had to make sure to match them correctly on JavaScript code. In particular, until ECMAScript 2015, JavaScript did not include syntactical support for classes. Classes were *emulated* using functions as constructors, and methods/fields are added to their prototype [46, 47, 48]. In addition, *object literals* could represent imaginary classes: comma-separated list of name-value pairs enclosed in curly braces, where the name-value pairs declare the class fields/methods. We have taken all these aspects into account during the assignment task, to avoid misclassifications.

Our analysis revealed that, in 88% of bugs in BUGSJS, the fix includes changes falling into one of the proposed categories. The most prevalent bug fix patterns involve changing an `if` statement (*i.e.*, modifying the `if` condition or adding a precondition), changing assignment statements, and modifying function call arguments (Table VI). The same three categories have been also found to be most recurring in Java code, but with a different ordering: Pan et al. [44] report that the most prevalent fix patterns are changes done on method calls, `if` conditions, and assignment expressions. In addition, we found that changes to class fields are also prevalent. This can be explained by the fact that in JavaScript, object literals are frequently created without the need for defining a class or function constructor, and, as far as fixing bugs is concerned, updating their attributes (*i.e.*, fields) is a common practice.

C. JavaScript-related Bug Patterns

We found three *new* recurring patterns in our benchmark, which we describe next.

Changes to the `return` statement's expression. We found a recurring bug-fixing pattern involving changing the return statement's expression of a function, *i.e.*:

```
- return node.type !== "A";
+ return !(node.type === "A" && lastI.type === "R");
```

Variable declaration. In JavaScript, it is possible to use a variable without declaring it. However, this has implications which might lead to subtle *silent bugs*. For example, when a variable is used inside a function without being declared, it is "hoisted" to the top of the global scope. As a consequence, it is visible to all functions, *outside its original lexical scope*, which can lead to name clashes. This fix pattern essentially includes declaring a variable which has already been in use.

Initialization of empty variables. This bug-fixing pattern category corresponds to Hanam et al.'s first bug pattern, *i.e.*, Dereferenced non-values. To avoid this type of bug, developers can add additional `if` statements, comparing values against "falsey" values ("`undefined`" type, or "`null`"). This bug fix pattern provides a shortcut to using an `if` statement, by using a logical "or" operator, *e.g.*, `a = a || {}`, which means that the value of `a` will remain intact if it already has a "non-falsey" value, or it will be initialized with an empty object otherwise.

V. Discussion

Overall, our analysis reveals that the bug fixes included in BugsJS cover a diverse range of categories, some of which being specific only to JavaScript. As such, these results might drive devising novel software analysis and repair techniques for JavaScript, with BugsJS being a suitable real world bug benchmark for evaluating such techniques.

In the next section, we discuss some of the possible use cases of our benchmark in supporting empirical studies in software analysis and testing, as well as its limitations and threats to validity of our study.

A. Possible Use Cases

1) Testing techniques: Various fields of testing research can benefit from BugsJS. First, our benchmark includes more than 25k JavaScript test cases, which makes it a rather large dataset for different regression testing studies (*e.g.*, test prioritization, test minimization, or test selection). Second, BugsJS can play a role to support research in software oracles (*e.g.*, automated generation of semantically-meaningful assertions), as it contains all test suites' evolution as well as examples of real fixes made by developers. Additionally, these can be used to drive the design of automated test repair techniques [49, 50]. Finally, test generation or mutation techniques for JavaScript can be evaluated on BugsJS at a low cost, since pre-computed coverage information are available for use.

2) Bug prediction using static source code analysis: To construct reliable bug prediction models, training feature sets are extracted from the source code, comprising instances of buggy and healthy code, and static metrics. BugsJS can support these studies since it streamlines the hardest part of constructing the training and testing datasets, that is, determining whether a given code element is affected by a bug. As such, the cleaned fixes included in BugsJS make this task much easier. Also, the availability of both uncleaned and cleaned bug-fixing patches in the dataset can allow assessing the sensitivity of the proposed models to the noise.

3) Bug localization: BugsJS can support devising novel bug localization techniques for JavaScript. Approaches that use NLP can take advantage of our benchamark since bugs are readily available to be processed. Indeed, text retrieval techniques are used to formulate a natural language query that describes the observed bug. To this aim, BugsJS contains pointers to the natural language bug description and discussions for several hundreds of real world bugs. Similarly, BugsJS will be of great benefit for other popular bug localization approaches, *e.g.*, the spectrum-based techniques [51, 52, 53, 54].

4) Automated program repair: Automated program repair techniques aim at automatically fixing bugs in programs, by generating a large pool of candidate fixes, to be later validated. 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.

B. Limitations

The initial version of BugsJS includes only server-side JavaScript applications developed with the Node.js framework. As such, experiments evaluating the client-side (e.g., the DOM) are not currently supported. While our survey revealed a large number of subjects being used for evaluating such techniques, the majority of these programs could not be directly included in our proposed benchmark.

Indeed, in the JavaScript realm, the availability of many implementations, standards, and testing frameworks poses major technical challenges with respect to devising a uniform and cohesive bugs infrastructure. Similar reasoning holds for selecting Mocha as a reference testing framework.

Running tests for browser-based programs may require complex and time-consuming configurations. When dealing with a large and diverse set of applications, achieving isolation would require automating each single configuration for all possible JavaScript development and testing frameworks, which can be a cumbersome task. Nevertheless, all the subjects included in BugsJS have been previously used by at least one work in our literature survey (*e.g.*, BOWER, SHIELDS, KARMA, NODE-REDIS, and MONGOOSE are all used in bug-related studies).

C. Threats to validity

The main threat to the *internal validity* of this work is the possibility of introducing bias when selecting and classifying the surveyed papers and the bugs included in the benchmark.

Our paper selection was driven by the keywords related to software analysis and testing for JavaScript (Section II). We may have missed relevant studies that are not captured by our list of terms. We mitigate this threat by performing an issue-by-issue, manual search in the major software engineering conference proceedings and journals, followed by a snowballing process. We, however, cannot claim that our survey captures all relevant literature; yet, we are confident that the included papers cover the major related studies.

Concerning the bugs, we manually classified all candidate bugs into different categories (Section III-C), and the retained bugs into categories pertaining to existing bug and fix taxonomies (Section IV-B). To minimize classification errors, multiple authors simultaneously analyzed the source code and performed the classifications individually, and disagreements were resolved by further discussions among the authors.

Threats to the *external validity* concern the generalization of our findings. We selected only 10 applications and our bugs may not generalize to different projects, or other relevant classes of bugs might be unrepresented within our benchmark.

We tried to mitigate this threat by selecting applications with different sizes and pertaining to different domains. However, other subject systems are necessary to fully confirm the generalizability of our results, and corroborate our findings.

With respect to *reproducibility* of our results, all classifications, subjects, and experimental data are available online, making the analysis reproducible.

VI. Related work

A. C, C++, and C# benchmarks

The Siemens benchmark suite [55] was one of the first datasets of bugs used in testing research. It consists of seven C programs, containing manually seeded faults. The first widely used benchmark of real bugs and fixes is the SIR (Software-artifact Infrastructure Repository) [14]. It contains multiple versions of Java, C, C++, and C# programs comprising test suites, bug data, and scripts. The benchmark contains both real and seeded faults, the latter being more frequent.

Le Goues et al. [16] proposed two benchmarks for C programs called ManyBugs and IntroClass,[25] which include 1,183 bugs in total. The benchmarks are designed to support the comparative evaluation of automatic repair, targeting large-scale production (ManyBugs) as well as smaller (IntroClass) programs. ManyBugs is based on nine open-source programs (5.9M LOC and over 10k test cases) and it contains 185 bugs. IntroClass includes 6 small programs and 998 bugs.

Rahman et al. [56] examined the OpenCV project mining 40 bugs from seven out of 52 C++ modules into the benchmark *Pairika*.[26] The seven modules analyzed contain more than 490k LOC, about 11k test cases and each bug is accompanied by at least one failing test.

Lu et al. [57] propose *BugBench*,[27] a collection of 17 open-source C/C++ programs containing 19 bugs pertaining to memory and concurrency issues.

B. Java benchmarks

Just et al. [15] presented Defects4J, a bug database and extensible framework containing 357 validated bugs from five real-world Java programs. BugsJS shares with Defects4J the idea of mining bugs from the version control history. However, BugsJS has some additional features: subject systems are accessible in the form of *git* forks on a central GitHub repository, which maintains the whole project history. Further, all programs are equipped with pre-built environments in form of Docker containers. Moreover, in this paper we also provide a more detailed analysis of subjects, tests, and bugs.

Bugs.jar [58] is a large-scale dataset intended for research in automated debugging, patching, and testing of Java programs. Bugs.jar consists of 1,158 bugs and patches, collected from eight large, popular open-source Java projects.

iBugs [59] is another benchmark containing real Java bugs from bug-tracking systems originally proposed for bug localization research. It is composed of 390 bugs and 197k LOC coming from three open source projects.

[25]http://repairbenchmarks.cs.umass.edu/

[26]https://github.com/tum-i22/Pairika

[27]https://github.com/lihebi/bugbench

QuixBugs [18] is a benchmark suite of 40 confirmed bugs used in program repair experiments targeting Python and Java with passing and failing test cases.

BugSwarm [17] is a recent dataset of real software bugs and bug fixes to support various testing empirical experiments such as test generation, mutation testing, and fault localization.

To our knowledge, BugsJS is the first benchmark of bugs and related artifacts (e.g., source code and test cases) that targets the JavaScript domain. In addition, BugsJS differentiates from the previously-mentioned benchmarks in the following aspects: (1) the subjects are provided as *git* forks with complete histories maintained, (2) a framework is provided with several features enabling more convenient usage of the benchmark, (3) the subjects and the framework itself are available as GitHub repositories, (4) Docker container images are provided for easier usage, and (5) the bug descriptions are accompanied by their discussions in natural language.

VII. Conclusions

The increasing interest of developers and industry around JavaScript has fostered a huge amount of software engineering research around this language. Novel analysis and testing techniques are being proposed every year, however, without a centralized benchmark of subjects and bugs, it is difficult to fairly evaluate, compare, and reproduce research results.

To fill this gap, in this paper we presented BugsJS, a benchmark of 453 real, manually validated JavaScript bugs from 10 popular JavaScript programs. Our quantitative and qualitative analyses show the diversity of the bugs included in BugsJS that can be used for conducting highly-reproducible empirical studies in software analysis and testing research related to, among others, regression testing, bug prediction, and fault localization for JavaScript. Using BugsJS in future studies is further facilitated by a flexible framework implemented to automate checking out specific revisions of the programs' source code, running each of the test cases demonstrating the bugs, and reporting test coverage.

As part of our ongoing and future work, we plan to include more subjects (and corresponding bugs) to the benchmark. 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.

Acknowledgments

Gyimesi and Vancsics were supported by project EFOP-3.6.3-VEKOP-16-2017-0002, co-funded by the European Social Fund. Beszédes was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. This work was partially supported by grant 2018-1.2.1-NKP-2018-00004 "Security Enhancing Technologies for the IoT" funded by the Hungarian National Research, Development and Innovation Office. Ministry of Human Capacities, Hungary grant 20391-3/2018/FEKUSTRAT is acknowledged. Mesbah, Stocco and Mazinanian were supported in part by NSERC Discovery and DAS grants.

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] M. Billes, A. Møller, and M. Pradel, "Systematic black-box analysis of collaborative web applications," in *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[3] J. Wang, W. Dou, C. Gao, Y. Gao, and J. Wei, "Context-based event trace reduction in client-side JavaScript applications," in *Proc. of International Conference on Software Testing, Verification and Validation (ICST)*, 2018.

[4] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei, "A comprehensive study on real world concurrency bugs in Node.js," in *Proc. of International Conference on Automated Software Engineering*, 2017.

[5] 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.

[6] M. Ermuth and M. Pradel, "Monkey see, monkey do: Effective generation of GUI tests with inferred macro events," in *Proc. of 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016.

[7] C. Q. Adamsen, A. Møller, R. Karim, M. Sridharan, F. Tip, and K. Sen, "Repairing event race errors by controlling nondeterminism," in *Proc. of 39th International Conference on Software Engineering (ICSE)*, 2017.

[8] M. Madsen, F. Tip, E. Andreasen, K. Sen, and A. Møller, "Feedback-directed instrumentation for deployed JavaScript applications," in *Proc. of 38th International Conference on Software Engineering (ICSE)*, 2016.

[9] S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Hybrid DOM-sensitive change impact analysis for JavaScript," in *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[10] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Transactions on Software Engineering*, vol. 37, no. 5, 2011.

[11] R. Gopinath, C. Jensen, and A. Groce, "Mutations: How close are they to real faults?" in *Proc. of International Symposium on Software Reliability Engineering*, 2014.

[12] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proc. of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[13] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. of International Conference on Software Engineering*, 2005.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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*.

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

[20] 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.

[21] Z. Gao, C. Bird, and E. T. Barr, "To type or not to type: quantifying detectable bugs in JavaScript," in *Proc. 39th International Conference on Software Engineering*, 2017.

[22] Q. Hanam, F. S. d. M. Brito, and A. Mesbah, "Discovering bug patterns in JavaScript," in *Proc. of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016.

[23] F. S. Ocariza Jr, K. Pattabiraman, and A. Mesbah, "Detecting unknown inconsistencies in web applications," in *Proc. of 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[24] F. S. Ocariza, Jr., K. Pattabiraman, and A. Mesbah, "Detecting inconsistencies in JavaScript MVC applications," in *Proc. of 37th International Conference on Software Engineering (ICSE)*, 2015.

[25] F. S. Ocariza, G. Li, K. Pattabiraman, and A. Mesbah, "Automatic fault localization for client-side JavaScript," *Softw. Test. Verif. Reliab.*, vol. 26, no. 1, Jan. 2016.

[26] F. S. Ocariza, Jr., K. Pattabiraman, and A. Mesbah, "Vejovis: Suggesting fixes for JavaScript faults," in *Proc. of 36th International Conference on Software Engineering*.

[27] J. Davis, A. Thekumparampil, and D. Lee, "Node.fz: Fuzzing the server-side event-driven architecture," in *Proc. of 12nd European Conference on Computer Systems (EuroSys)*, 2017.

[28] A. M. Fard and A. Mesbah, "JavaScript: The (un)covered parts," in *Proc. of IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[29] S. Mirshokraie and A. Mesbah, "JSART: JavaScript

assertion-based regression testing," in *Web Engineering (ICWE)*, 2012, pp. 238–252.

[30] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient JavaScript mutation testing," in *Proc. of 6th International Conference on Software Testing, Verification and Validation (ICST)*, 2013.

[31] ——, "Guided mutation testing for JavaScript web applications," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 429–444, May 2015.

[32] ——, "Atrina: Inferring unit oracles from GUI test cases," in *Proc. of International Conference on Software Testing, Verification and Validation (ICST)*, 2016.

[33] ——, "JSEFT: Automated JavaScript unit test generation," in *Proc. of 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015.

[34] C. Quist, G. Mezzetti, and A. Møller, "Analyzing test completeness for dynamic languages," in *Proc. of International Symposium on Software Testing and Analysis*.

[35] A. M. Fard, A. Mesbah, and E. Wohlstadter, "Generating fixtures for JavaScript unit testing," in *Proc. of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[36] S. Artzi, J. Dolby, S. H. Jensen, A. Moller, and F. Tip, "A framework for automated testing of JavaScript web applications," in *33rd International Conference on Software Engineering (ICSE)*, 2011.

[37] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Transactions on Software Engineering*, 2012.

[38] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu, "A survey of dynamic analysis and test generation for JavaScript," *ACM Comput. Surv.*, vol. 50, no. 5, pp. 66:1–66:36, Sep. 2017.

[39] S. Hong, Y. Park, and M. Kim, "Detecting concurrency errors in client-side JavaScript web applications," in *Proc. of IEEE 7th International Conference on Software Testing, Verification and Validation*, 2014.

[40] M. Dhok, M. K. Ramanathan, and N. Sinha, "Type-aware concolic testing of JavaScript programs," in *Proc. of 38th International Conference on Software Engineering*, 2016.

[41] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. of 27th International Conference on Software Engineering*.

[42] E. Giger, M. Pinzger, and H. C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Proc. of 8th Working Conference on Mining Software Repositories (MSR)*, 2011, pp. 83–92.

[43] E. C. Campos and M. d. A. Maia, "Common bug-fix patterns: A large-scale observational study," in *Proc. of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017.

[44] 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.

[45] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proc. of 37th International Conference on Software Engineering (ICSE)*, 2015, pp. 913–923.

[46] L. H. Silva, M. T. Valente, and A. Bergel, "Refactoring legacy JavaScript code to use classes: The good, the bad and the ugly," in *Mastering Scale and Complexity in Software Reuse*, 2017.

[47] S. Rostami, L. Eshkevari, D. Mazinanian, and N. Tsantalis, "Detecting function constructors in JavaScript," in *Proc. of IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016.

[48] L. Eshkevari, D. Mazinanian, S. Rostami, and N. Tsantalis, "JSDeodorant: Class-awareness for JavaScript Programs," in *Proceedings of the 39th International Conference on Software Engineering Companion*, 2017.

[49] A. Stocco, R. Yandrapally, and A. Mesbah, "Visual web test repair," in *Proc. of 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018.

[50] M. Hammoudi, G. Rothermel, and A. Stocco, "WATERFALL: An incremental approach for repairing record-replay tests of web applications," in *Proc. of 24th International Symposium on Foundations of Software Engineering (FSE)*, 2016.

[51] 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.

[52] A. Perez, R. Abreu, and A. van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches," in *Proc. of 39th International Conference on Software Engineering (ICSE)*, 2017.

[53] 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.

[54] A. Perez, R. Abreu, and M. D'Amorim, "Prevalence of single-fault fixes and its impact on fault localization," in *Proc. of IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[55] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proc. of 16th International Conference on Software Engineering*, 1994.

[56] M. R. Rahman, M. Golagha, and A. Pretschner, "Pairika: A failure diagnosis benchmark for C++ programs," in *Proc. of 40th International Conference on Software Engineering: Companion*, 2018.

[57] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[58] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.Jar: A large-scale, diverse dataset of real-world Java bugs," in *Proc. of 15th International Conference on Mining Software Repositories (MSR)*, 2018.

[59] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proc. of International Conference on Automated Software Engineering*.