Bugfox: A Trace-Based Analyzer for Localizing the Cause of Software Regression in JavaScript

Yuefeng Hu
The University of Tokyo
Bunkyo, Japan
huyuefeng99@csg.ci.i.u-tokyo.ac.jp

Hiromu Ishibe
The University of Tokyo
Bunkyo, Japan
ishibe@csg.ci.i.u-tokyo.ac.jp

Feng Dai
The University of Tokyo
Bunkyo, Japan
daifeng@csg.ci.i.u-tokyo.ac.jp

Tetsuro Yamazaki
The University of Tokyo
Bunkyo, Japan
yamazaki@csg.ci.i.u-tokyo.ac.jp

Shigeru Chiba
The University of Tokyo
Bunkyo, Japan
chiba@acm.org

Abstract

Software regression has been a persistent issue in software development. Although numerous techniques have been proposed to prevent regression from being introduced before release, few are available to address regression as it occurs post-release. Therefore, identifying the root cause of regression has always been a time-consuming and labor-intensive task. We aim to deliver automated solutions for solving regressions based on tracing. We present Bugfox, a trace-based analyzer that reports functions as the possible cause of regression in JavaScript. The idea is to generate runtime trace with instrumented programs, then extract the differences between clean and regression traces, and apply two heuristic strategies based on invocation order and frequency to identify the suspicious functions among differences. We evaluate our approach on 12 real-world regressions taken from the benchmark *BugsJS*. First strategy solves 6 regressions, and second strategy solves other 4 regressions, resulting in an overall accuracy of 83% on test cases. Notably, Bugfox solves each regression in under 1 minute with minimal memory overhead (<200 Megabytes). Our findings suggest Bugfox could help developers solve regression in real development.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; Software maintenance tools.

Keywords: Regression, Debugging, Runtime Tracing, Code Transformation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SLE '24, October 20–21, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1180-0/24/10
https://doi.org/10.1145/3687997.3695648

ACM Reference Format:
Yuefeng Hu, Hiromu Ishibe, Feng Dai, Tetsuro Yamazaki, and Shigeru Chiba. 2024. Bugfox: A Trace-Based Analyzer for Localizing the Cause of Software Regression in JavaScript. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE '24), October 20–21, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3687997.3695648

1 Introduction

Software regression, where previously functional software exhibits new faults after updates, remains a persistent challenge in software development. While many techniques focus on preventing regressions before release [5], fewer tools effectively address regressions post-release [18]. Identifying the root cause of these regressions can be a tedious and time-consuming task for developers, often requiring extensive manual debugging. This issue is particularly pronounced in complex codebases with extensive commit histories and numerous interconnected components.

In this paper, we introduce a trace-based analyzer designed to automate the localization of the root causes of software regressions in JavaScript. Our approach is grounded on the premise that discrepancies in execution traces between a properly functioning base program and a faulty, updated program can reveal the root cause of regressions. Based on this idea, we develop Bugfox, which traces both base and faulty programs by instrumenting their source code, extracts the differences of two traces, and reports suspicious functions among the differences based on two heuristic strategies. The first strategy focuses on the invocation relationships of function calls, aiming to identify the starting point of deviated behavior of the buggy program and return that one function call as the root cause of regression. The second strategy examines the frequency of function calls inside the differences of two traces, and returns top four functions with most occurrences as candidates. Bugfox collects those candidates and reports five candidates in total as possible root causes of regression.

We evaluate Bugfox on 12 real-world regressions from the BugsJS [6] benchmark, demonstrating its effectiveness and efficiency. The first strategy successfully resolves 6 regressions, and the second strategy resolves an additional 4 regressions, achieving an overall accuracy of 83% across test cases. Notably, Bugfox was able to solve each regression in under one minute with minimal memory overhead, making it a practical tool for real-world development environments.

Our research contribution involves an detailed exploration of identifying the root cause of regression through tracing in real-world scenarios. This includes an implementation of this methodology in JavaScript, along with the construction of 12 regression cases derived from an open-source dataset. Our research artifacts, including source code and constructed dataset, are available at Zenodo [8]. These artifacts allow for the reproducibility of our experimental results.

2 Localize the Cause of Regression

Regression has long been a persistent problem in software development process. Although detecting the regression is well studied in previous works [21, 22], localizing the cause of regression is still under exploration. In Figure 1, we show an example of a regression from a real-world project. The regression is from Hessian.js [3], which is the JavaScript implementation of Hessian binary web service protocol [10]. In Hessian.js, there are many utility functions implementing the encoding of raw data and decoding of binary data based on Hessian protocol. This example involves the release version 2.1.9 *(2b53d13)* (referred as the base version) and its subsequent commit *(a46fbc9)* (referred as the upgraded version). The green code with a minus - symbol is from the base version, and the red code with a plus + symbol is from the upgraded version.

The example implicitly introduces a regression in the upgraded version by incorrectly decoding the binary when raw data is `null` object. As shown in Figure 1, the condition of the `if` statement at *line 11* is a regular expression that matches everything except strings with the word `this` followed by a dollar sign `$` and then one or more digits at the end of the string, such as `this$123` or `this$4567`. Therefore, the program would enter the `if` branch when `key` is a `null` object, and it would be encoded to an object with property `null` and value `'null'`, which is the correct implementation based on Hessian protocol when dealing with `null`. The upgraded version mistakenly adds a precondition to the `if` statement, leading to the program only entering the `if` branch when `key` is a string or number, and the `null` object would not be encoded in that scenario. Besides, the initial `if` statement at *line 3* relates to the new feature introduced to the upgraded version, which involves supporting `java.util.HashMap` to ES6 map. It assigns a `$map` property to object `result.$` in this code snippet. To side with these modifications in Figure 1, the upgraded version also changes its corresponding unit testing function of decoding `null` object as illustrated in Figure 2, to pass all testing modules. Since the upgraded commit *(a46fbc9)* also involves new features and dozens of file changes, the developer merges this commit to the main tree without carefully reviewing this small change. This commit is blamed by other developers as a regression later and gets rolled back in a later fix *(37d19e7)*.

In most cases, addressing regressions still relies heavily on manual debugging by developers rather than relying on automated tools. This regression example may be perceived as straightforward to resolve, but imagine the regression is discovered after a long period of time along with numerous commits covering on the "regression" one, localizing the root cause of regression would become greatly difficult. In addition, in this example `proto.readObject` is just a middleware utility function used for implementing the decoding in hessian.js, while `hessian.decode` relates to hundreds of functions which all could possibly be regarded as the root cause of regression when developers inspect the limited output of failures reported by testing framework such as Mocha [4]. Given the complexity of pinpointing regression among extensive commits and intricate codebases, automated tools become indispensable and time-saving for efficiently resolving such tasks.

Among all testing techniques in modern test-driven development, unit testing as the most commonly used technique, still has limitations on addressing regressions and is not satisfying. Unit testing is also known as module testing, where individual units or components of a software application are tested in isolation to ensure they function correctly as standalone units. However, with the tremendous expansion of modern software scale, it is considered burdensome and

```
1  proto.readObject = function (withType) {
2    ...
3  + if (isMap) {
4  +   Object.defineProperty(result.$, '$map', {
5  +     value: new Map(),
6  +     enumerable: false,
7  +   });
8  + }
9    this._addRef(result);
10   ...
11 - if (!/^this\$\d+$/.test(key) {
12 + t = typeof key;
13 + if ((t === 'string' || t === 'number') &&
14 +  !/^this\$\d+$/.test(key) {
15     result.$[key] = value;
16   }
17   ...
18 }
```

Figure 1. Base (green) and upgraded (red) "buggy" version of function `proto.readObject` in decoder.js.

```
1 describe('map.test.js', function () {
2   it('should decode successful when key
       is null', function () {
3     var data = new Buffer([77, 116, 0,
         0, 78, 83, 0, 4, 110, 117, 108,
         108, 122]);
4     var rv = hessian.decode(data);
5 -   rv.should.eql(null: 'null');
6 +   rv.should.eql();
7   });
8 }
```

Figure 2. Base and upgraded "buggy" version of testing function related to foregoing `proto.readObject` function.

unpractical to design testing modules and test cases for every single unit and boundary condition; accordingly, testing modules always includes unit that directly or indirectly invokes other units, and regression often causes the infection of failures in this situation where one failed unit may affect the behavior of other units. As similar circumstance arises in integration testing, this chain reaction might result in vast failed test cases and bring greater difficulty to developers to locate the root cause of regression.

3 Bugfox

In this paper, we present Bugfox [8], which is a trace-based analyzer for localizing the cause of software regression in JavaScript. Our approach is based on the speculation that the root cause of regression could be detected inside the differences between execution traces of base program and buggy program, where the former represents the program that functions properly with a given unit test and the latter represents the updated program but fails on the same unit test. On the basis of this idea, we develop Bugfox, which traces both base and buggy programs by instrumenting their

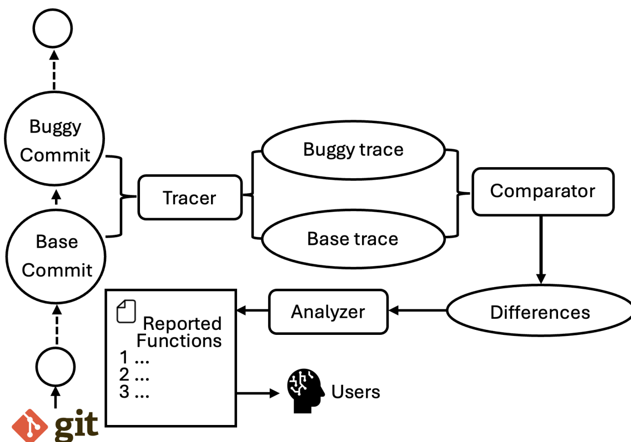

Figure 3. Framework of Bugfox

source code, extracts their differences, and reports suspicious functions among the differences based on two heuristic strategies. Bugfox serves as a local assistant for developers encountering regressions after modifying the source code but before pushing their changes to the repository. It aids in resolving regressions locally, acting as a pre-commit and pre-push diagnostic tool. In addition, Git, as the state-of-the-art version control system, is responsible for maintaining and switching the version of source code in our approach. We assume that a buggy program is also committed, probably, to a debugging branch. So we below call base and buggy programs *base commit* and *buggy commit*, respectively.

We give an overview of the workflow of Bugfox in Figure 3. The workflow could be divided into three independent process: tracing, comparing and analyzing. Section 3.1 to 3.3 describes their specification.

3.1 Runtime Tracing in JavaScript

As illustrated in Figure 3, the inputs of this workflow include two versions of source code, namely the base commit and the buggy commit. The base commit functions correctly, while the buggy commit introduces a regression in certain module. Usually, the buggy commit resides chronologically after the base commit on the Git tree. Bugfox traces the execution of these programs and generates call trace [2] separately, namely a tree that represents the calling relationships between various functions within a program. Each node in the call trace represents a single function call, and the edges between nodes indicate the flow of invocation relationships between functions. An edge connects a node f to a node g when a function f is invoked, and a function g is invoked within the body of that function f.

A call-trace node has several attributes. One is an identifier to identify a called function. It is a concatenation of the path name of a source file, the names of functions surrounding the definition of the called function. For example, let function `onfinish` be defined in function `sendfile` in file `lib/response.js`. The identifier is this character string:

```
lib/response.js#Func@sendfile/Func@onfinish
```

Here, # and / are separators. `Func@` represents the kind of the function. BugFox supports seven kinds of functions: `Func` (normal function), `FuncVar` (function variable), `FuncExpr` (function expression), `method` (method in a class), `PropFunc` (property function), `ArrowMethod` (special arrow function as a method in a class), and `AnonFunc` (anonymous function). When a function does not have a name, the hash value of its body is used as its name.

Besides an identifier, a call-trace node has other attributes. It contains the hash value of the body of a called function, function arguments (the values of function parameters), the `this` object, and a return value. The value of `this` and the values of function parameters are obtained and recorded as node attributes twice, before and after the execution of the

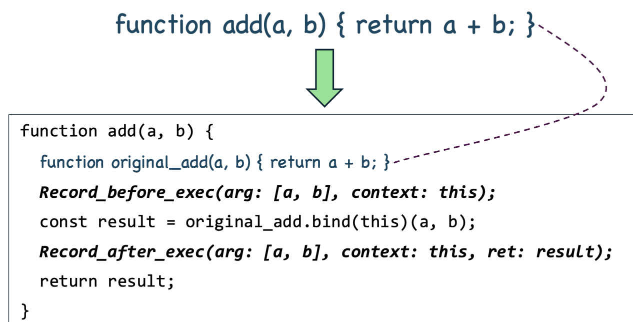

Figure 4. Code transformation of a function definition (*italic* text represents pseudo-code for a tracing statement).

function body. These values are represented by character strings produced by the built-in function `JSON.stringify` with our extension to support function objects, circular objects, objects with accessors, and other objects that are not well-supported by the original `stringify`. When computing the hash value of a function body, the function definition is first transformed into a canonicalized form to absorb differences due to formatting.

Bugfox generates call traces for a base commit and a buggy commit taken from a git repository. For this aim, Bugfox conducts code transformation, which instruments the original source code in JavaScript to insert tracing statements that record runtime values as execution trace during program execution. Figure 4 illustrates an example of code transformation. After code transformation, the instrumented program is run to record execution trace in memory. The recorded execution trace is finally written in a file before the program execution finishes.

3.2 Differences Extraction between Call Traces

After obtaining the call traces from tracing, we need to extract the ***differences*** between the base and the buggy traces. A difference is hereby defined as a pair of matching function node coming from two call traces, but differing in the value of *arguments*, *this* object and *return value*. In order to accurately match up the homologous function node from two call traces, we set up four prerequisites to guarantee two nodes represents same function call in their programs: 1) Same path (from root node) in the call traces, 2) Both are n-th child node of same parent node, 3) Node with same function name, 4) All their previous sibling nodes are matched under same rules. Rule 1 and 2 guarantee two function calls have same invocation relationship in their program. Rule 3 ensures they stands for same function, so that we can analyze their differences legally. Last rule express that once two nodes are not matched, we stop the traversal of its remain sibling nodes since the unmatch indicates the invocation path inside its parent function execution has been altered, therefore we cannot guarantee the remaining sibling nodes represents the same function call even if satisfy the first three prerequisites. Through this matching approach, we are capable of identifying the first deepest differing nodes, considered as the starting point of deviated behavior in the buggy program.

After matching function nodes from two call traces in depth first order (DFS), we compare their value of arguments, this object and return value before and after its execution. If any of them is different, we mark this pair of function node as a *difference* between two traces. In case these values are different before its execution, it indicates this deviation of its behavior might be caused by its caller (passing different arguments or changing the context of one function call) instead of itself. Similarly, if these values are different only after its execution, it signifies the function itself is responsible for the different behavior due to its code modification or inner function calls. Furthermore, we compare the hash value of their function definitions to judge whether the function is being modified in the buggy commit. These results of comparison, including whether function specifications are being changed before its execution, and whether being changed after its execution, and whether function definition is modified in buggy commit, are collected for later analysis.

3.3 Localization of the Root Causes

Finally, Bugfox localizes the root causes of regression by analyzing the differences extracted in the previous step. It adopts two heuristic strategies to prioritize the most likely candidates among the pairs of call-trace nodes marked as different. One strategy focuses on the invocation order of function calls and selects one candidate. The other strategy focuses on the frequency of function calls inside the differences, and it selects up to four candidates if feasible. Notably, these strategies may overlap, potentially resulting in a maximum total of five candidates presented as potential root causes of regression.

First deepest function (FDF). The first heuristic strategy identifies the *first deepest function* (FDF) among the differences, aiming to identify the starting point of deviated behavior of the buggy program. *Deepest functions* are the node pairs that are marked as different but do not have descendants marked as different in their subtrees. The function calls represented by those pairs show deviated behavior, but the nested calls to other functions from those calls do not show deviated behavior. We consider those *deepest function* calls as possible *root* causes of regression. Among all those deepest function calls, the chronologically first one could be regarded as the starting point of deviation of the buggy program, and thus it could be also considered as the beginning of the regression.

Therefore, Bugfox selects the first deepest function (FDF) call as a candidate of the cause of regression. Before reporting it as a candidate, however, Bugfox checks how the selected pair of nodes is different from each other. As listed in Table 1,

Table 1. The decision table for the first deepest function strategy.

Case	Is code changed	Is input state changed	Is output state changed	Speculation	Final reported function
1	no	no	yes	Deviated behavior inside its execution	itself
2	yes	no	yes	Regard *code modification* as the cause of deviated behavior inside its execution	itself
3	no	yes	no	Rare condition where input state varies while output state remains the same	caller
4	yes	yes	no	Rare condition where input state varies while output state remains the same	caller
5	no	yes	yes	Possibly caused by the different arguments or `this` object before execution	caller
6	yes	yes	yes	Most complicated situation, need further inspection on the detail of difference	itself

Bugfox compares the attributes of the nodes of that pair. It compares the hash value of function bodies to check whether the source code changes between the base and buggy programs. It also compares the arguments (the values of the function parameters) and `this` object before and after the function execution. In Table 1, the input states are those values before function execution while the output states are those values after function execution. If the results of the comparison match case 3, 4, or 5, then Bugfox reports that a candidate is the parent of the first deepest function call, which is the function that invokes the first deepest function call. For example, when a function f calls another function g and the call to g is the first deepest, the function f is a candidate of the cause of regression. If the results of the comparison match the other cases 1, 2, or 6, then Bugfox reports that a candidate is the function invoked by the first deepest function call. For the example above, the function g is a candidate.

Top-n. The other heuristic strategy counts the occurrences of each function in the differences, then ranks them, and reports the top-n functions as candidates of the cause of regression. If multiple functions share the same count, the function with the earliest invocation is being selected. Each pair of nodes marked as different has an identifier as an attribute. This identifier represents the name of a called function. Bugfox checks this attribute to count the calls to each function. We assume that a function called in more node pairs marked as different is the cause of regression with a greater likelihood.

Bugfox generally uses 4 as n for the Top-n strategy. It reports the top-4 functions as candidates of the cause of regression. We choose 4 to balance the accuracy and efficiency of this strategy. It is more likely that the selected candidates include the true cause of regression when n is larger. However, the users of Bugfox must spend more time to investigate which candidate is a true cause. Notably, when n is selected, the outcomes for n ranging from 1 to $n-1$ are also inherently included due to the ranking by counts. For instance, in Table 6, if n equals 1 is selected, only the top candidate will be reported, and so forth.

Case study. We now revisit the Hessian.js example discussed in section 2, using it as a case study to illustrate how these two heuristic strategies work in practice. In this example, the modification to function `proto.readObject` in `lib/v1/decoder.js` is responsible for the regression. After comparing two call traces, Bugfox identifies 12 pairs of matching function nodes with differing behavior, referred as *differences* in this paper, while the ground truth is also included in the differences. Detailed experimental result of this example is presented as Hessian Bug-2 in Table 6.

In *FDF* strategy, function node `proto._addRef` is recognized as the first deepest function, referred to *line 9* in Figure 1. It is a utility function defined as:

```
function (obj) { this.refMap[this.refId++] = obj; }
```

The arguments before execution in 1) base version and 2) buggy version are:

```
1) {"$class":"java.util.HashMap","$":{}}
2) {"$class":"java.util.HashMap","$":{"$map":{}}}
```

These arguments remain unchanged after the execution. The difference about `$map` property directly points to the code at *line 3* in Figure 1, indicates the starting point of deviation relates to the modification on `proto.readObject`. The result also indicates that the source codes of `proto._addRef` are identical between two versions, while the arguments *before* and *after* its function execution vary between two

versions. Therefore, this example corresponds to case 5 in Table 1, and Bugfox reports its callers `proto.readObject` as the final candidate of this strategy, which matches the ground truth. Furthermore, when developer checks the specification of its caller `proto.readObject` in the call traces, the return value in base version `{"null":"null"}` and buggy version `{"$map":{}}` explicitly demonstrates that the original functionality of decoding `null` object is partly ignored and affected by the modification at *line 11* in Figure 1. These findings suggest that Bugfox would be instrumental in addressing the regression in this example.

In *Top-n* strategy, Bugfox returns 4 candidates, includes function `proto.read` with 3 times, `exports.decode`, `proto.readObject` and `proto._addRef` with 1 time. If multiple functions share the same count, the one invoked earlier in the program is given higher priority. As a result, the ground truth `proto.readObject` is reported as the third candidate in this strategy. Since all candidates, except `exports.decode`, are defined in `lib/v1/decoder.js`, these findings can help developers quickly recall the modifications to these functions and fix the regression in this example.

3.4 Limitation

Our methodology highly depends on the *differences* extracted from two call traces. Predictably, whether two call traces match well and whether valuable differences are collected directly affects the accuracy of our heuristic strategies. There are plenty situations might lead to the above dilemma, the most significant of which is *large scale refactoring*. Big scale refactoring always concerns the addition or deletion of functions and huge change of invocation order of function calls. In that case, one call trace might contain function calls that the other one does not have, and the matching of functions would be difficult due to the change of invocation order. Thereupon according to the rule of matching two call traces, function calls which related to those refactorings would not be legally extracted into the differences. The tool can hardly solve the regression if those refactorings are responsible for the regression.

In addition, there are scenarios where our tool would fail even the desired *differences* are collected. Depending on the functionality of programs, there exists function calls related to *"mutable"* information such as port number, IP address, timestamp, or other randomly generated data. Differences extracted from such function calls might impede the analysis of our heuristic strategies in several ways. As for FDF strategy, once these differences appears chronologically earlier than the actual answer, FDF strategy might return these unrelated functions as the answer. In Top-n strategy, in case the frequency of such differences is higher than the actual answer, these unrelated functions will be ranked ahead the actual answer in *n*-candidates. Since these unrelated functions might result in false positive in our approach, we consider them as *noisy functions*.

4 Experiments

In this paper, we propose two research questions as follows.

RQ1. Can Bugfox be used to localize the root causes of regression?

RQ2. Can Bugfox meet the performance demands in real-world development?

In order to answer these research questions, we evaluate our system on 12 real-world regressions extracted from a JavaScript bug dataset *BugsJS* [6]. In this section, we will introduce our experiment settings, experiment results and the answers to the RQs. The experiments are conducted on a machine with an *11th Gen Intel(R) Core(TM) i7-11700 @ 2.50GHz* CPU and 64GB RAM, and the operating system is Ubuntu 20.04.4 LTS.

4.1 Dataset and Experiment Settings

Currently, there is a lack of dataset especially focusing on regression bugs. In order for better evaluation, we screen and extract 12 regressions from a famous JavaScript bug dataset called *BugsJS* [6], which is a benchmark containing 453 real and manually-validated JavaScript bugs from 10 popular JavaScript server-side projects [12].

In order to extract these regressions, we take an automated approach to identify regression bugs systematically and efficiently, instead of manually inspecting each bug to determine if it is a regression. In *BugsJS*, each bug corresponds to a real Git commit and is labeled with a unique ID, its bug description, and its fix in real world that includes the update of the unit tests and source code. To note, the labeled Git commit may not include the cause of the bug, and it simply states the occurrence or detection of bug. To determine if a bug is a regression, we traverse the Git tree in reverse chronological order starting from the commit labeled as a bug, and execute the *updated* unit tests on every preceding commits. If we ever get a test failure in a commit followed by a test success in its parent commit, the bug could be considered as a regression since the same functionality has been correctly implemented before the code stops working. Meanwhile, we mark the commit that firstly introduces the bug as the *regression commit*.

We successfully recognize 12 bugs in *BugsJS* that could be considered as regressions [8]. These bugs will be used to evaluate the accuracy and performance of Bugfox in this paper. We collect their regression commits, and identify the functions being updated in a later fix as the root cause of the regressions. These functions are also considered as the ground truth to be compared with the results from Bugfox. In detail, 7 of these regression bugs come from Express [7], 3 of them come from ESLint [24], and 2 of them come from Hessian.js [3]. We give an overview of the collected bugs in Table 2. Note that the unit test of each bug case fails, and the generated log messages show only the failed assertion and the call trace of its test unit. Testing frameworks give

Table 2. List of 12 regressions extracted from *BugsJS*.

Project	Bug-ID	Regression commit	Unit testing module
Express	1	f41d09a	test/app.options.js
Express	8	cf41a8f	test/app.use.js
Express	9	997a558	test/app.js
Express	13	31b2e2d	test/app.param.js
Express	16	c6e6203	test/res.redirect.js
Express	18	fb2d918	test/app.param.js
Express	27	7f04916	test/app.param.js
ESLint	10	a21dd32	tests/lib/config.js
ESLint	134	5266793	tests/lib/rules/no-useless-escape.js
ESLint	307	0f97279	tests/lib/rules/no-multi-spaces.js
hessian.js	2	a46fbc9	test/map.test.js
hessian.js	8	29f434e	test/v1.test.js

no reasonable analysis or speculation for users to further understand and solve the regression. Finding the causes of regression is difficult without an extra tool like Bugfox.

4.2 RQ1: Can Bugfox be Used to Localize the Root Causes of Regression?

In each regression in Table 2, Bugfox will report 5 potential functions in total as the possible root causes of the regression, including one from FDF strategy and four from Top-n strategy. If the reported function aligns with the ground truth, we consider it as a success. Table 3 summarizes the results of applying Bugfox to localize the root cause of regression on the previously mentioned dataset. The 3rd column shows the fixed function for each regression in real world, considered as the ground truth in our experiment. The 4th column indicates whether the function reported by FDF strategy matches the ground truth. The 5th column shows whether the ground truth is included in the top-4 candidates reported by Top-n strategy with $n = 4$, and if included, which candidate matches the ground truth. Checkmark symbol ✓indicates the result of the strategy matches the expected result, while blank cell indicates the opposite.

As shown in Table 3, FDF strategy solves 6 regressions out of 12, and Top-n strategy solves 8 regressions out of 12. Among them, 4 regressions are shared by both strategies. In FDF strategy, all 7 regressions from Express meet the case 5 in Table 1, which return the caller of first deepest function as the root cause. Hessian.js Bug-2 meets the case 5 and returns the caller, and remaining 4 regressions meet the case 2 and return the first deepest function itself. By the integration of two strategies, Bugfox covers a total of 10 regressions, resulting in an overall accuracy of 83.3%.

We further verify the limitations of Bugfox in real scenario by investigating the failures of our experiments. It can be seen that the results of FDF strategy on Express project are not ideal. After inspecting the details of these cases, we find that Express, as a *de facto standard* server framework for Node.js, contains a massive amount of utility or middle-ware functions that include *unrelated* information such as port numbers, timestamps, IP addresses, or other randomly generated data. Such noise is liable to cause different behavior between two versions, leading to undesired differences of traces being collected and analyzed before the real suspicious difference. This situation happens in many regressions from Express project, resulting in the false reports of FDF strategy on Bug-9, 13, 16, 18, 27 from Express project. It also results in the failure of Top-n strategy on Bug-16 from Express project. However, Top-n strategy shows more tolerance since it provides more than one candidate.

Particularly, the complete failure of Bug-1 from Express project relates to the large-scale refactoring: the function that introduces the regression is added to the buggy commit for the first time. Hence, the base trace does not contain that function, resulting in the inability to extract meaningful differences and resolve the regression effectively.

4.3 RQ2: Can Bugfox Meet the Performance Demands in Real-World Development?

To evaluate the performance of our system, we record the running time of different stages in Bugfox, the lines of code (LOC) of each test project, and the size of tracing log of entire program. The size of tracing log is used as the approximation of memory overhead.

In terms of code transformation, Table 4 demonstrates the lines of code (LoC) of each regression and their corresponding transformation time. It can be seen that the regressions from ESLint project spend more time that other regressions. This is because that ESLint, as a static code analysis tool for identifying problematic patterns found in JavaScript code, contains a huge numbers of unit tests, resulting in thousands of functions being transformed in runtime. However, since code transformation is a one-time task in our workflow and all such transformations spend only a few seconds, we argue that the time of code transformation is acceptable in real usage of this system.

Table 5 demonstrates the overall performance of Bugfox on memory and runtime, aiming to evaluate the overhead of applying tracing in our system. The 3rd column and 4th column show the runtime of the original program and instrumented program. The comparison of these two columns illustrates the runtime overhead of tracing. It can be seen that the overhead varies widely in our experiment. For example, in Bug-2 of hessian.js, the runtime of the instrumented program is only 1.3 times larger than the original program, while in Bug-8 of Express, the ratio increases to 2400 times. After inspecting the detail of traces, we speculate that this discrepancy is caused by different amount of data being traced and time spent on their object serialization. However, the

Table 3. Experimental results of Bugfox on 12 regression test cases.

Project	Bug-ID	Expected result	FDF strategy	Top-n strategy with n=4
Express	1	lib/router/index.js#FuncExpr@proto.handle		
	8	lib/application.js#FuncExpr@app.use	✓	2nd candidate
	9	lib/application.js#FuncExpr@app.use		3rd candidate
	13	lib/router/index.js#FuncExpr@proto.process_params		4th candidate
	16	lib/response.js#FuncExpr@res.redirect		
	18	lib/router/index.js#FuncExpr@proto.process_params		1st candidate
	27	lib/router/index.js#FuncExpr@proto.process_params		4th candidate
ESLint	10	lib/config.js#Class@Config/Method@constructor	✓	1st candidate
	134	lib/rules/no-useless-escape.js#PropFunc@create	✓	
	307	lib/rules/no-multi-spaces.js#PropFunc@create/PropFunc@Program	✓	
hessian.js	2	lib/v1/decoder.js#FuncExpr@proto.readObject	✓	3rd candidate
	8	lib/utils.js#FuncExpr@exports.handleLong	✓	3rd candidate

maximum runtime of instrumented program in our experiment takes less than 26 seconds, which could be considered acceptable in real-world development process. The 5th column shows the time spent on analyzing differences between traces, and in all cases, the maximum analysis time is less than 10 seconds. The last column demonstrates the size of traces stored in the disk, as an indicator to evaluate the memory overhead of tracing, it varies from 7MB to 132MB. In our experiment, our system spends less than 1 minute for solving a regression with memory overhead less than 200MB for all cases. Compared with manual debugging, Bugfox observably saves time for our users and meet our expected performance demands.

Table 4. Lines of code and code transformation runtime.

Project	Bug-ID	Lines of code	Code transformation runtime
Express	1	7211	1.145 s
Express	8	10189	1.426 s
Express	9	9631	1.393 s
Express	13	7630	1.153 s
Express	16	9653	1.403 s
Express	18	8309	1.231 s
Express	27	8306	1.206 s
ESLint	10	222470	7.331 s
ESLint	134	169489	6.422 s
ESLint	307	225086	7.335 s
hessian.js	2	4805	1.016 s
hessian.js	8	4339	950.017 ms

Table 5. Overall performance of Bugfox on memory and runtime.

Project	Bug-ID	Original runtime	Runtime of instrumented program	Analysis time of differences	Size of trace
Express	1	12 ms	3 s	527.086 ms	28 MB
Express	8	12 ms	18 s	277.583 ms	125 MB
Express	9	2 ms	3 s	1.807 s	41 MB
Express	13	12 ms	18 s	4.591 s	93 MB
Express	16	11 ms	26 s	6.457 s	132 MB
Express	18	10 ms	15 s	1.633 s	80 MB
Express	27	13 ms	23 s	8.085 s	99 MB
ESLint	10	24 ms	35 ms	162.41 ms	24 MB
ESLint	134	15 ms	2 s	1.136 s	7 MB
ESLint	307	18 ms	236 ms	265.122 ms	8 MB
hessian.js	2	3 ms	4 ms	197.577 ms	39 MB
hessian.js	8	5 ms	33 ms	165.294 ms	41 MB

4.4 Threats to Validity

There are several threats to internal validity of our evaluation. The number of regression cases used in our experiment is very limited, which may lead to bias of experimental results. Further benchmarking with different test frameworks is needed to comprehensively evaluate our system. Moreover, the use of heuristic strategies in localizing the suspicious functions also reduces the generalizability of Bugfox. In particular, the selection of value n (number of candidates) in Top-n strategy is highly ad-hoc. Further research is required to analyze the setting of this parameter.

As explained in the failures of Bug-1 and Bug-16 in Express, our current approach has limitations on dealing with regressions with large scale refactoring or noisy functions, as they are lead to great difficulty on matching two call traces and extracting usable differences for later analysis. Furthermore, as shown in Table 5, the overhead of tracing varies a lot across different regression cases. Although Bugfox can

work smoothly on tested modules with small granularity, the overhead of complete tracing would be enormous on middle-level tested modules or integration testing.

5 Related Work

Pastor et al. [19] proposes a tool for debugging regression problems in C/C++ software, which uses GDB debugger to trace predicate and then reports a list of suspicious differences in the base and upgraded version. However, due to the limitation of low-level procedural programming languages, their approach is unable to implement universal tracing with complete data serialization, and only a small subset of program is applied when identifying the cause of regressions. To validate the feasibility of utilizing universal tracing for addressing regressions, we develop our trace-based analyzer in JavaScript, which captures a comprehensive range of program information.

Maksimovic et al. [15] presents a framework that utilizes traditional machine learning techniques along with historical data in version control systems and the results of functional debugging. Their approach aims to rank revisions based on their likelihood of being responsible for a particular failure. This research motivates the construction of a regression bug dataset based on the version control system in this paper.

Unit testing frameworks have been implemented for various languages and platforms, including C [23], Java [1], JavaScript [9], Python [13], .NET [17, 20], Node.js [4, 11, 14] and React [16]. These frameworks are widely used in modern software development process, providing the possibility to extend our approach to other programming languages.

Rosero [21] presents a survey of software regression testing techniques applied from 2000 to 2015, considering their application domain, the types of metrics they use, their application strategies and the phases of the software development process where they are applied. Our approach is inspired by its survey on regression test selection (RTS) techniques, where it describes the trend of testing techniques based on graphs, dependency relationships, historical data and heuristics.

Table 6. Detailed experimental results of Bugfox on 12 regression test cases.

Project	Bug-ID	Expected result	FDF strategy	Top-n strategy with n=4
Express	1	lib/router/index.js# FuncExpr@proto.handle	lib/application.js# FuncExpr@app.enable (Case 5)	lib/application.js#FuncExpr@app.set (4) lib/application.js#AnonFunc@d7d4b9c/FuncExpr@app[method] (4) lib/application.js#FuncExpr@app.enable (2) lib/express.js#Func@createApplication (1)
Express	8	lib/application.js# FuncExpr@app.use	lib/application.js# FuncExpr@app.use (Case 5)	lib/utils.js#FuncExpr@exports.flatten (2) lib/application.js#FuncExpr@app.use (1) lib/application.js#FuncExpr@app.use/AnonFunc@497a64c (1) lib/router/index.js#FuncExpr@proto.use (1)
Express	9	lib/application.js# FuncExpr@app.use	lib/application.js# FuncExpr@app.enable (Case 5)	lib/application.js#FuncExpr@app.set (46) lib/application.js#FuncExpr@app.enabled (6) lib/application.js#FuncExpr@app.use (5) lib/application.js#FuncExpr@app.lazyrouter (5)
Express	13	lib/router/index.js# FuncExpr@proto.process_params	lib/application.js# FuncExpr@app.enable (Case 5)	lib/application.js#FuncExpr@app.set (11) lib/application.js#FuncExpr@app.enabled (3) lib/application.js#AnonFunc@4c5011c/FuncExpr@app[method] (3) lib/router/index.js#FuncExpr@proto.process_params (3)
Express	16	lib/response.js# FuncExpr@res.redirect	lib/application.js# FuncExpr@app.enable (Case 5)	lib/application.js#FuncExpr@app.set (12) lib/router/index.js#FuncExpr@proto.match_layer (3) lib/router/index.js#FuncExpr@proto.process_params (3) lib/router/layer.js#FuncExpr@Layer.prototype.handle_request (3)
Express	18	lib/router/index.js# FuncExpr@proto.process_params	lib/application.js# AnonFunc@4c5011c/FuncExpr@app[method] (Case 5)	lib/router/index.js#FuncExpr@proto.process_params (4) lib/application.js#FuncExpr@app.set (2) lib/application.js#FuncExpr@app.handle (1) lib/application.js#AnonFunc@4c5011c/FuncExpr@app[method] (1)
Express	27	lib/router/index.js# FuncExpr@proto.process_params	lib/application.js# FuncExpr@app.enable (Case 5)	lib/application.js#FuncExpr@app.all/AnonFunc@68e1170 (33) lib/router/route.js#AnonFunc@cea325e/FuncExpr@Route.prototype[method] (33) lib/application.js#FuncExpr@app.set (13) lib/router/index.js#FuncExpr@proto.process_params (4)
ESLint	10	lib/config.js# Class@Config/Method@constructor	lib/config.js# Class@Config/Method@constructor (Case 2)	lib/config.js#Class@Config/Method@constructor (1) lib/config/plugins.js#Class@Plugins/Method@constructor (1)
ESLint	134	lib/rules/no-useless-escape.js# PropFunc@create	lib/rules/no-useless-escape.js# PropFunc@create (Case 2)	lib/util/comment-event-generator.js#Func@emitComments (40) lib/util/comment-event-generator.js#Func@emitCommentsEnter (20) lib/util/comment-event-generator.js#Func@emitCommentsExit (20) lib/code-path-analysis/code-path-analyzer.js#Func@forwardCurrentToHead (18)
ESLint	307	lib/rules/no-multi-spaces.js# PropFunc@create/PropFunc@Program	lib/rules/no-multi-spaces.js# PropFunc@create/PropFunc@Program (Case 2)	lib/testers/rule-tester.js#Class@RuleTester/Method@run/Func@testValidTemplate (1) lib/testers/rule-tester.js#Class@RuleTester/Method@run/Func@runRuleForItem (1) lib/linter.js#Class@Linter/Method@verify (1) lib/util/traverser.js#Class@Traverser/Method@traverse (1)
hessian.js	2	lib/v1/decoder.js# FuncExpr@proto.readObject	lib/v1/decoder.js# FuncExpr@proto.readObject (Case 5)	lib/v1/decoder.js#FuncExpr@proto.read (3) index.js#FuncExpr@exports.decode (1) lib/v1/decoder.js#FuncExpr@proto.readObject (1) lib/v1/decoder.js#FuncExpr@proto._addRef (1)
hessian.js	8	lib/utils.js# FuncExpr@exports.handleLong	lib/utils.js# FuncExpr@exports.handleLong (Case 2)	test/v1.test.js#AnonFunc@52d7217/AnonFunc@3df64bc/AnonFunc@fa86bb7/AnonFunc@efe399a (1) lib/v1/decoder.js#FuncExpr@proto.readLong (1) lib/utils.js#FuncExpr@exports.handleLong (1) lib/v1/decoder.js#FuncExpr@proto.handleType (1)

6 Conclusion

Addressing software regression heavily relies on manual debugging in software development, automated tools are needed to increase the efficiency of regression testing. We propose Bugfox towards this goal, based on inserting instrumentation to trace complete program execution, and localize the root cause of regression among differences of clean and regression programs. We explore the performance of Bugfox on 12 test case studies. FDF strategy solves 6 regressions and Top-n strategy solves other 4 regressions, resulting in an overall accuracy of 83%. Eventually, Bugfox spends less than 1 minute with minimal memory overhead less than 200 megabytes. Our results show that Bugfox is able to help developers solve regressions in real development. Our future work is to investigate how to automatically identify and collect the regressions in open-source projects and retest our tool on a more comprehensive benchmark. Another future work is reducing overheads due to obtaining execution trace in JavaScript.

Acknowledgments

This work is partly supported by the JSPS KAKENHI Grant Numbers JP20H00578 and JP24H00688.

A Experimental Details

Table 6 shows the detailed experimental results of Bugfox on 12 regression test cases. The 3rd column shows the ground truth of the root cause of regression. The 4th column shows the function reported from FDF strategy and indicates which case it fits in Table 1. The 5th column shows top four candidates reported from Top-n strategy with n=4 in descending order of frequency, where the number in parentheses indicates the frequency of each candidate. Reported function in green color indicates that it matches the expected result.

References

[1] Kent Beck, Erich Gamma, David Saff, and Kris Vasudevan. 2024. JUnit - a programmer-friendly testing framework for Java and the JVM. https://junit.org/junit5/.

[2] D. Callahan, A. Carle, M.W. Hall, and K. Kennedy. 1990. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering* 16, 4 (1990), 483–487. https://doi.org/10.1109/32.54302

[3] Yiyu He Fengmk and Others. 2023. hessian.js - a js hessian binary web service protocol. https://github.com/node-modules/hessian.js.

[4] OpenJS Foundation. 2024. Mocha - the fun, simple, flexible test framework. https://mochajs.org/.

[5] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. 2001. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.* 10, 2 (apr 2001), 184–208. https://doi.org/10.1145/367008.367020

[6] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. 2019. BugsJS: a Benchmark of JavaScript Bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 90–101. https://doi.org/10.1109/ICST.2019.00019

[7] TJ Holowaychuk, StrongLoop, et al. 2024. Express - fast, unopinionated, minimalist web framework for Node.js. https://expressjs.com/.

[8] Yuefeng Hu, Hiromu Ishibe, Feng Dai, Tetsuro Yamazaki, and Shigeru Chiba. 2024. *Artifact - Bugfox: A Trace-based Analyzer for Localizing the Cause of Software Regression in JavaScript*. https://doi.org/10.5281/zenodo.13814132

[9] Cypress.io Inc. 2024. cypress - fast, easy and reliable testing for anything that runs in a browser. https://www.cypress.io/.

[10] Caucho Technology Inc. 2022. Hessian binary web service protocol. http://hessian.caucho.com/.

[11] Developer Express Inc. 2024. TestCafe - a Node.js tool to automate end-to-end web testing. https://testcafe.io/.

[12] Hiromu Ishibe. 2023. *A cause detector of software regressions by comparing program execution traces*. Master's thesis. The University of Tokyo, Japan. https://csg-www.s3.ap-northeast-1.amazonaws.com/public/papers/23/master-ishibe.pdf

[13] Pekka Klärck, Janne Härkönen, et al. 2024. Robot - generic automation framework for acceptance testing and RPA. https://robotframework.org/.

[14] Pivotal Labs. 2024. jasmine - simple JavaScript testing framework for browsers and node.js. https://jasmine.github.io/.

[15] Djordje Maksimovic, Andreas Veneris, and Zissis Poulos. 2015. Clustering-based revision debug in regression verification. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*. 32–37. https://doi.org/10.1109/ICCD.2015.7357081

[16] Christoph Nakazawa, Orta Therox, Simen Bekkhus, and Ricky. 2024. Jest - a delightful JavaScript testing framework with a focus on simplicity. https://jestjs.io/.

[17] James Newkirk and Brad Wilson. 2024. xUnit - a free, open source, community-focused unit testing tool for .NET. https://xunit.net/.

[18] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 199–209. https://doi.org/10.1145/2001420.2001445

[19] Fabrizio Pastore, Leonardo Mariani, and Alberto Goffi. 2013. RADAR: A tool for debugging regression problems in C/C++ Software. In *2013 35th International Conference on Software Engineering (ICSE)*. 1335–1338. https://doi.org/10.1109/ICSE.2013.6606711

[20] Charlie Poole, James Newkirk, Alexei Vorontsov, Michael Two, Philip Craig, Rob Prouse, Simone Busoli, and Neil Colvin. 2024. NUnit - a unit-testing framework for all .NET languages. https://nunit.org/.

[21] Raúl H. Rosero, Omar S. Gómez, and Glen Rodríguez. 2016. 15 Years of Software Regression Testing Techniques — A Survey. *International Journal of Software Engineering and Knowledge Engineering* 26, 05 (2016), 675–689. https://doi.org/10.1142/S0218194016300013

[22] Hema Srikanth and Myra B. Cohen. 2011. Regression testing in Software as a Service: An industrial case study. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 372–381. https://doi.org/10.1109/ICSM.2011.6080804

[23] Mark VanderVoord, Doctor Surly, et al. 2024. unity - simple unit testing for C. https://www.throwtheswitch.org/unity.

[24] Nicholas C. Zakas. 2024. ESLint - a tool for identifying and reporting on patterns found in ECMAScript/JavaScript code. https://eslint.org/.

Received 2024-07-01; accepted 2024-08-30