Auto-Grading Dynamic Programming Language Assignments

Liang Gong
University of California, Berkeley
gongliang13@eecs.berkeley.edu

ABSTRACT

One challenge for massive online education is to develop automatic grading system that provide useful feedback. We present a new method for auto-grading programming assignments of introductory JavaScript Courses. Our system takes as input the student's solution and instructor's reference implementation for concrete testing and profiling. Based on the collected information, the system automatically finds counter examples revealing bugs in given solution and provide feedback including grading, bad code practise as well as recommendation on bug locations.

The system uses symbolic execution technique to achieve high test coverage and statistical bug localization technique to pinpoint bugs. Program analysis technique is heavily used in our system to detect cheating, prevent infinite loops and detect inefficient code patterns. We have evaluated our system on real-world course assignments obtained from the Introduction to Data Structure course at UC Berkeley (CS61B). Our results show that our system can locate on average 60% of bugs in the top 5 lines of code recommended. With the help of our system, we also pinpoint bugs in publicly available code repository.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*symbolic execution, testing tools*

General Terms

Formal methods, Symbolic Execution, Bug Localization

Keywords

Dynamic Analysis; JavaScript; Bugs

1. INTRODUCTION

JavaScript is becoming increasing popular. For web programming, increasing computations are migrated to font-end web page to provide real-time service and smooth user experience. Many important applications (*e.g.,* Facebook, Gmail, Google Map *etc.*) are written in JavaScript. While other traditional programming languages such as Java and even C/C++ starts to compile their code into JavaScript so that their applications can run in a browser. So JavaScript is the *de facto* web assembly language. The rise of server-side JavaScript on Node.js also takes on more responsibility for back-end computation.

At the same time, massive open online courses (*abbr.* MOOC) attracts a lot of interests and attentions in providing high quality education to people worldwide. MOOC provides increasing opportunities for students and instructors across the world and thus attracts more and more people registering online courses. As a result, providing feedback for the enormous amount of homework submissions emerges as a challenge in the online classroom. In courses where the student-teacher ratio can be very high, it is impossible for instructors to personally give feedback to students which leads to a new research problem:

> *With the increasing number of student's programming assignments submission, can we automatically grade the student's solution while providing useful feedback?*

In this paper, we present an automated technique to provide feedback for JavaScript introductory programming assignments. Our approach is a test-based technique that adopts symbolic execution technique to find counter examples and achieves high test coverage. Based on the runtime information collected from high coverage testing, the approach analyses those information to automatically recommend possible program bug locations to provide students with feedback about how the program goes wrong and where it should be changed to fix the bug.

The problem of providing feedback appears to be related to the problem of bug localization, but they are different mainly due to the presence of the *reference implementation* which is the correct code provided by the instructor. In conventional bug localization research, only 1) buggy program, 2) a set of test cases and 3) corresponding test output (*test oracles*) are given. Due to the absence of correct implementation, collecting test oracles are considered expensive as they are manually labelled by developers[1]. This limits the massive application of trace-based bug localization. While for auto-grading problem on the other hand, it is safe to assume that the reference implementation is always is available from the instructor and thus the bug localization technique can collect test oracles by runing the reference implementation with low extra cost.

[1]This is known as the test oracle issue.

CS294-98 UC Berkeley, Spring'14

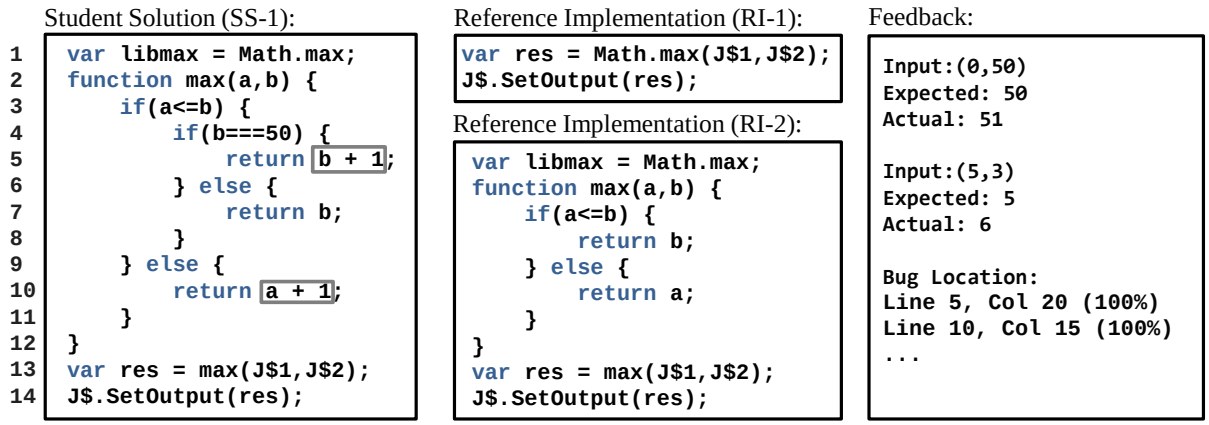

Figure 1: A Running Example.

We show the effectiveness of our approach by running our system on course assignments from real-world course as well as publicly available implementation of classical algorithm and data structures (*e.g.,* JavaScript implementation of quick sort on Github[2]). Then we use classical bug localization evaluation metric to measure the accuracy of our bug localization system.

The main contributions of this paper are as follows:

- Different from existing test-based auto-grading systems, we proposed a auto-grading approach based on concrete testing that can achieve high test coverage rate and recommend possible bug locations. Our system also solves some of the practical issues of concrete testing based auto-grading system (*e.g.,* cheating, and non-terminate programs).
- Our approach accepts reference implementation which may be a system call (without function source code). Existing formal method based auto-grading systems requires not only source code of reference implementation but also additional specification (*e.g.,* error model[5]) to assist reasoning on bug locations. Our approach is based on statistical analysis on runtime information that does not requires those detailed specifications and collecting which might be an extra burden on the users.
- Our implementation supports almost all of the advanced program constructs (*e.g.,* higher order functions etc.) in JavaScript. existing auto-grading system for high level languages(*e.g.,*[5] for Python) often only supports limited programming constructs.
- We have evaluated our approach on assignments from real-world courses and publicly available code snippets. The evaluation demonstrates that our system can effectively find counter examples and generate feedbacks to assist auto-grading and online education.

The rest of this paper is organized as follows: Section 2 shows a running example of our system. Section 3 gives an overview of our auto-grading system. The following sections present the key building blocks of our system, which includes program instrumentation, symbolic execution, statistical bug localization *etc.* Section 11 shows the empirical evaluation and Section 12 describes related work. Finally, Section 13 concludes this paper.

[2] `https://github.com/`

2. RUNNING EXAMPLE

Figure 1 shows an running example of our auto-feedback system which shows what the input and output are required by our system . The user of our system provides 1) *Student Solution (SS-1)* and 2) *Reference Implementation* of either form *RI-1* or *RI-2*. In this example, we use our system to auto-grade a `max` function implemented by the student (*i.e.,* SS-1). The student's implementation contains two bugs: 1) on line 5 when $(b \geq a) \wedge (b = 50)$, instead of returning `b`, the buggy program returns `b+1`; 2) on line 10 when $b < a$ instead of returning `a`, the buggy program returns `a+1`. The reference implementation accepted by our system can be either (RI-1): calling the library function `Math.max` provided by the JavaScript native library; or (RI-2): a correct implementation of `max` function.

Our system will automatically generate test cases to achieve high testing coverage and find counter example (if any) to trigger bugs in the student solution, and recommend possible bug location (see *feedback* in Figure 1). Note that the student may cheat by invoking the library function $Math.max$, our proposed system also detects calling forbidden functions (configurable by the instructor). At line 1 of student's solution, although the program reads the forbidden function, it was never invoked. So our system will not consider it as cheating and thus will not generate any warning. Static checking techniques can not accurately classify this kind of "cheating" in complicated program structure due to the limitation of alias analysis.

As mentioned, existing approaches can generally be classified into two categories: 1) auto-grading based on formal methods [40] and test-based online judge platform [9]. Formal methods based approaches often require an error model to assist bug localization which leads to an extra manual effort. Moreover, when the source code of the reference implementation is unavailable (*e.g.,* RI-1 in Figure 1), the formal methods cannot do reasoning to locate bugs. Existing test-based online judge approaches often require users to manually provide a set of test cases and corresponding

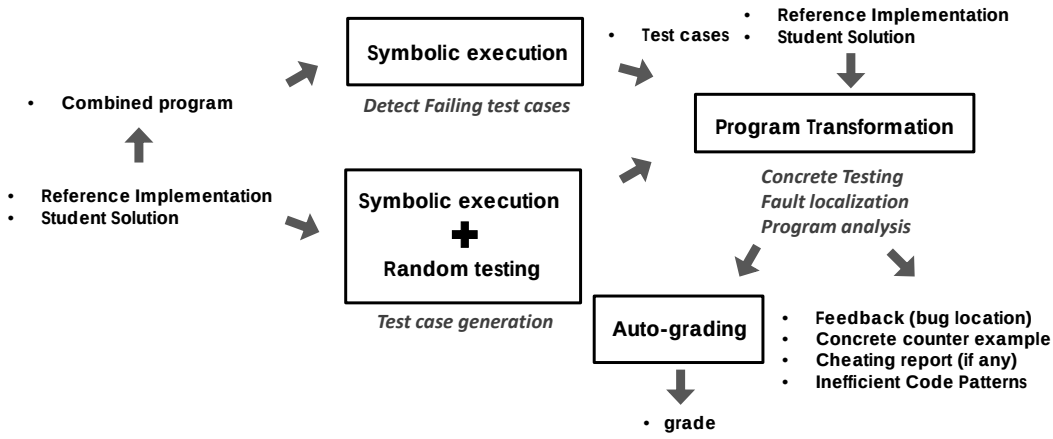

Figure 2: Approach Overview.

test output (*test oracle*). Providing fixed test cases cannot achieve high test coverage and thus often misses bugs in the student's solution. Furthermore, this test-based platform often only generate a final score based on the ratio of test cases passed and do not provide feedbacks like bug locations.

In contrast, our system does not require either error model [5] or test cases and test oracles [9]. Our system also automatically provides counter example that triggers the bugs and recommend possible bug locations to facilitate manual debugging. The following sections will introduce our approach in detail.

3. PROPOSED APPROACH

Figure 2 shows the key building blocks of the proposed auto-grading system. The arrow flow indicates the data flow and the order in which each components are activated. In the beginning, when the *Student Solution* and *Reference Implementation* are provided by the user, our proposed system will use *test case generation* techniques including *symbolic execution*(section 6) and *random testing*(section 5) to generate a specified number of test cases for concrete testing. To enhance the probability that our system detect a abnormal behaviour when there is a bug in the student solution, our proposed system adopts a technique to generate a counter example to trigger bugs (see section 9). Once all test cases are collected, we use program instrumentation (section 4) to profile the program execution under those generate test cases, the system also execute the exact same test input on the reference implementation to collect test oracles. Once test oracles and runtime information (*i.e.,* program spectra) are collected, we perform statistical fault localization (section 8) and get possible bug locations in the student's solution. During the execution of the instrumented program, our system monitors the execution to detect cheating, infinite loops and inefficient code patterns (section 10). Finally the bug location, counter examples, cheating information, inefficient code patterns (if any) and final grading are gathered and report to the user. In the following sections, we briefly introduce the rational and working mechanism behind those key components.

4. DYNAMIC ANALYSIS FRAMEWORK

Dynamic program analysis is the analysis of software applications by running the programs with a set of test inputs and analysing their behaviours during executions. This technique is widely used for software testing, software profiling, understanding software behaviours and many other applications of interest.

Conventionally, implementing a dynamic analysis framework for a dynamic programming language modifies the underlying virtual machine [35] to intercept and monitor low-level events. This approach however, has the following disadvantages:

Not all virtual machines are open source.

Some JavaScript virtual machines are not open source, or have strong license, which prohibits access to the source code or modifying the virtual machine for any purpose freely.

Hard to maintain the analysis framework code.

JavaScript virtual machine vendors has no obligation of preserving the inner function/method interface of the virtual machine. Consequently the previous developing effort are in vain if the dynamic analysis framework depends on inner functions which are removed or modified in the future.

Different programming paradigms.

Modifying the JavaScript virtual machines makes it difficult for a third-party developer to build the analysis module. JavaScript developers might not be familiar with the C/C++ programming paradigm and does not necessary have the background knowledge and experience of handling low level mechanism in which a glitch may crash the system.

In contrast, source code transformation makes the hooks and analysis code completely decoupled with the low level virtual machine and addresses all of the above difficulties. Further more, manipulating the source code makes the framework easily portable to any virtual machine that implements ECMAScript standard [13]. (*e.g.,* Mozilla SpiderMonkey [41], TraceMonkey [42] and Rhino [34], Nitro [15], Google V8 [14] used by Chrome and Node.js [16]*etc.*).

Moreover, our goal is doing dynamic program analysis on the fly in the browser. So the proposed approach instruments the program rather than modify VMs.

To illustrate the concept of code instrumentation, consider the following code snippet:

```
1 var a = b + c;
```

After instrumentation, the program is transformed into:

```
1 var a = W(B(R(b, 'b'), R(c, 'c'), 'b', 'c'), 'a');
```

Function `R(b, 'b')` means the callback function (*i.e.,* hook) that monitors the reading operation of variable `b`, the parameters of the callback function include the variable name and value. Similarly callback function `W` and `B` are for variable write and binary operations respectively. Inside each of those functions we will implement the semantics of the original JavaScript code and call an additional function stub:

```
1 function W(value, name) {
2     if(stub.write exists)
3         value = stub.write(value, name)
4     return value;
5 }
```

When executing the instrumented code, it not only performs the original semantics but also calls those stub function as dynamic programming analysis interfaces. But This is just a very simple case for the ease of understanding. JavaScript is a very flexible programming language with many dynamic programming language features, this requires our framework to be able to instrument many other programming constructs such as object/function/regexp/array literals, condition, loop, method/function call *etc.*

Despite the difficulties of implementing the code instrumentation for JavaScript, the benefits of doing code transformation are many-folds 1) it provides finer granularity of dynamic analysis interface; 2) no modification to the virtual machine is required; 3) analysis code is written in JavaScript which is familiar to front-end developers; 4) the JavaScript analysis framework is highly portable, as long as the JVM adheres to the ECMAScript standard [13], the JavaScript analysis framework can work on it with trivial migration effort.

As we adopt source manipulation to add hooks instead of modifying the underlying virtual machine and consequently the front-end analysis framework fuses analysis code into the programming context of the target code. The analysis code is executed in an external function to be called during the execution of the transformed target code. So the analysis code programming context and paradigm is the same as the JavaScript application. This makes it easy for any JavaScript developer to write their own analysis module. In contrast existing dynamic analysis framework often requires an understanding of the underlying virtual machine or physical machine mechanism, which is error-prone and demands a shift of programming paradigm.

5. RANDOM TESTING

Random testing is a black-box software testing technique in which test inputs are generated randomly. In Figure 1, `J$1` and `J$2` in student's solution (SS-1) reads randomly generated inputs and `J$.setOutput()` records the program output under the corresponding input. Same inputs are sent to reference implementation (RI-1 or RI-2) and `J$.setOutput()` records the reference implementation's output as the test oracle to determine whether the test passes or fails.

Although random testing generally works well for most programs, some bugs can hide in branches that may rarely be reached by random testing. For example, in the following listing, if we draw value for variable `a` and `b` randomly from an integer pool ranges from -100000 to 100000, the probability that we generate a test case that enters then branch would be $\frac{1}{200000}$.

```
1 if(a == 232) {
2     // error !;
3 }
```

So we need a more advanced technique to help enter those corner cases and branches.

6. SYMBOLIC EXECUTION

Symbolic execution[19, 24] is a kind of abstract interpretation technique that, instead of computing concrete values during the program execution, calculates intermediate values using logic formulas. Traditional static symbolic execution (*abbr.* SSE) interprets the program statically [24]. This usually cannot achieve high coverage, as the program may often turn into higher order logic (*e.g.,* higher order functions are turned into first order logic) or contain entities (*e.g.,* objects) that the SMT solver[11] is incapable of handling. Dynamic symbolic execution (*abbr.* DSE) techniques [8, 20, 38, 7] were proposed to address this issue. DSE executes the program symbolically when the generated formula can be handled by SMT solver, when it encounter entities or operations that leads to SMT solver out of theory, DSE uses concrete values instead. After executing the program once, symbolic execution technique generates a path constraint consisting of all path predicates collected along executing the programs. In order to generate a new test case that covers a previously unseen path. DSE negates the last path constraint and send the mutated path constraint into the SMT solver. The solver solves the path constraint and create a new test case that will lead to a execution covering other parts of the program under test.

To achieve better test coverage than existing system [9], we implement our own dynamic symbolic execution engine for JavaScript programs. Our DSE engine works in a similar way as KLEE [7] except 1) KLEE is implemented on top of LLVM [25] framework while our symbolic execution adopts program instrumentation and dynamic program analysis (section 4) to perform abstract interpretation and thus enables generating test cases; and 2) our implementation handles advanced programming features in JavaScript to achieve higher test coverage.

7. HANDLE ADVANCED FEATURES

Conventional symbolic execution are performed on either assembly or low level programming language (C/C++), so there is a limited kind of instructions and statements to handle by the symbolic execution. However, JavaScript is a dynamic programming language incorporating many new features which further widen the gap between the programming language semantics and the SMT solver's theory.

As an example, Figure 3 shows a simple program (in the left-upper side) with three possible branches. Existing symbolic execution tools will be able to handle this case properly as the path predicate is within the theory of SMT solvers. However, another simple program (in the upper-right side) with only two branches is beyond the capability of the existing symbolic execution engines. As the program

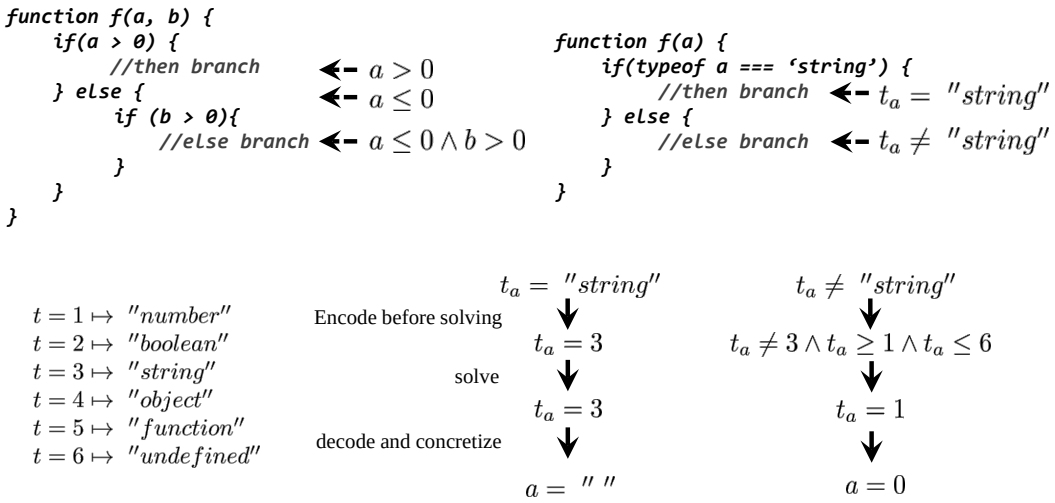

Figure 3: Handle `typeof` Operator in JavaScript.

uses unary operator `typeof` which returns a string representing the type of the operand. The `typeof` operator could return one of six values: `number`, `string`, `object`, `function`, `undefined` and `boolean`.

In order to handle this case, we extend the symbolic execution engine so that when it reaches the branch conditional expression `typeof a === 'string'` it generates a fresh symbolic variable $typeof\_a$ corresponding to the type of variable `a`. The existing SMT solver is capable of generating concrete input leading to the then branch as solving the constraint $typeof\_a =' string'$ is within its supported theory, and we next decode the constraint solver output to randomly generate a concrete input of type `string`.

The tricky part is generating the input leading to the `else` branch. As the path constraint turns into $typeof\_a \neq' string'$. A SMT solver may generate a random string as the solution which is meaningless. In order to constrain the output of SMT solver, we first decode all typeof symbolic values into integers according to the mapping relation shown in Figure 3 (in the left-down corner). For example, `string` type is mapped to integer 3 and $typeof\_a \neq' string'$ is encoded into $typeof\_a \neq 3$. In order to constrain the solution of the SMT solver in a meaningful domain, before each SMT query is sent to the SMT sovler, for each typeof constraint variable $typeof\_x$ we added an additional constraint: $typeof\_x \geq 1 \wedge typeof\_x \leq 6$. So now the encoded path constraint turns into:

$$typeof\_x \neq 3 \wedge typeof\_x \geq 1 \wedge typeof\_x \leq 6$$

For this query, the SMT solver generates the solution: $typeof\_x = 1$ followed by a post-processing step in our engine which decodes $typeof\_x = 1$ into $typeof\_x =' number'$ according to the map shown in Figure 3. Our symbolic execution engine then accordingly generates a concrete input of type `number` which guides the concrete execution to the `else` branch to achieve full test coverage.

Auto-grading is handled according to the ratio of test cases that fails. Some auto-grading system considers syntactic similarity as one factor for auto-grading deterministic finite automaton (DFA) assignments [3]. Here we only considers ratio of test cases that fails, as syntactically correct programming implementations to the same problem can be very different from each other. Besides, in our system reference implementation can simply invoke native function calls where source code are not available (*e.g.,* RI-1 in Figure 1).

8. STATISTICAL BUG LOCALIZATION

Spectrum-based fault localization(or statistical fault localization) techniques find suspicious faulty program entities in programs by comparing passed and failed executions.

Spectrum-based fault localization aims to locate faults by analysing program spectra of passed and failed executions. A program spectra often consists of information about whether a program element (e.g., a function, a statement, or a predicate) is hit in an execution. Program spectra between passed and failed executions are used to compute the suspiciousness score for every element. All elements are then sorted in descending order according to their suspiciousness for developers to investigate. Empirical studies (e.g., [29, 22]) show that such techniques can be effective in guiding developers to locate faults. Parnin *et al.* conduct a user study [32] and show that by using a fault localization tool, developers can complete a task significantly faster than without the tool on simpler code. However, fault localization may be much less useful for inexperienced developers.

Before we start looking at the fault localization, several concepts need to be stated. A **program spectrum**[3] [2] is a collection of data that provides a specific view on the dynamic behavior of software. This data is collected at runtime, and typically consist of a number of counters or flags for the different parts of a program.

In figure 4, each line represents a test case execution trace, while each column means a specific program entity(statement, block, or function)'s hit count in all test cases. The fault localization techniques rank each feature in the matrix according to their relevant score to the execution result. It is actually a feature selection on program spectrum.

The key for a spectrum-based fault localization technique

[3]Figure 4 is from from [2], because their explaination is accurate and better.

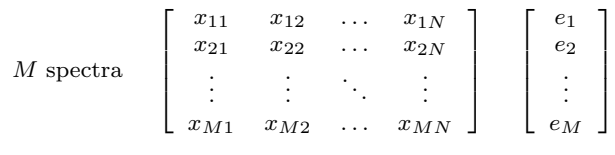

Figure 4: An illustration of Program Spectra.

is the formula used to calculate suspiciousness. The existing state-of-arts in fault localization fields are Jaccard[26], Tarantula[23] and Ochiai[26] etc. Zhang et al. summarizes 33 existing fault-localization techniques and compare their effectiveness on seimens benchmark dataset[12]. Table 1 lists the formulae of three well-known techniques: *Tarantula* [22], *Ochiai* [1], and *Jaccard* [1]. Given a program element s, $N_{ef}(s)$ is the number of ***failed*** executions that ***execute*** s; $N_{np}(s)$ numerates ***passed*** executions that do ***not*** hit s; by the same token, $N_{nf}(s)$ counts ***failed*** executions that do ***not*** hit s and $N_{ep}(s)$ counts ***passed*** executions that ***execute*** s.

Table 1: Spectrum-based fault localization

Name	Formula
Tarantula	$\frac{\frac{N_{ef}(s)}{N_{ef}(s)+N_{nf}(s)}}{\frac{N_{ef}(s)}{N_{ef}(s)+N_{nf}(s)} + \frac{N_{ep}(s)}{N_{ep}(s)+N_{np}(s)}}$
Ochiai	$\frac{N_{ef}(s)}{\sqrt{(N_{ef}(s)+N_{nf}(s)) \cdot (N_{ef}(s)+N_{ep}(s))}}$
Jaccard	$\frac{N_{ef}(s)}{N_{ef}(s)+N_{nf}(s)+N_{ep}(s)}$

Example. N_{ef}, N_{ep}, N_{nf}, and N_{np} can thus be calculated from the spectra. The suspiciousness scores of *Tarantula*, *Ochiai*, and *Jaccard* for each statement are then calculated based on the formulae in Table 1.

9. GENERATE COUNTER EXAMPLE

Although symbolic execution and random testing generates test cases that cover branches and paths, high coverage does not always guarantee finding a bug triggering test input. Without counter examples, statistical bug localization approaches can not pinpoint bugs as there is no behaviour difference between the buggy program and reference implementation. To avoid this potential issue, we adopt a technique for equivalence checking to find counter examples[39]. Specifically, given student solution p_s and reference implementation p_r, we check the semantics equivalence of the two programs by combining with the template:

```
p_c = function (input) {
    out1 ← p_s(input);
    out2 ← p_r(input);
    if (out1 ≠ out2) error;
}
```

The combined program p_c will be executed through the symbolic execution engine, if a test case entering the last then branch is feasible, that is an counter example as the outputs of the two programs (p_s and p_r) are different under the same input.

10. PRACTICAL ISSUES

Our system adopts dynamic program analysis based on concrete testing which allows us to collect runtime behaviour information for providing feedback, but also brings some unique issues comparing with statically checking the program. The following sections describe some of the practical issues for building a usable system based on dynamic program analysis and how we deal with them.

10.1 Detect Cheating

Invoking native library as reference implementation can save instructor's effort. However, that also means student can cheat by invoking those native library. For example, if the programming assignment is implementing a sorting algorithm, the student's submission may simply contains a statement calling `array.sort` which is a library method integrated in most JavaScript environment.

To address this issue, our program instrumentation framework rewrite function calling expression to intercept every function or method call to enable additional checking. For this issue, beforing calling a function/method f, our system checks if $f \in S_f$ which is a set of forbidden functions configurable by the instructor.

10.2 Kill Infinite Loops

Student solution may contain any form of bugs, including a loop that may never ends. For concrete testing based approach, that means evaluating the student's solution may leads to a frozen browser. Our system addresses this issue by instrumenting conditional expression in `while` and `for` loops to check if the conditional expression always evaluates to *true* for certain times. If the count is above a predefined threshold, the dynamic analysis terminates the loop by assigning the conditional expression a $false$ value. A warning on infinite loop also be reported.

10.3 Detect Inefficient Code Patterns

Another key building block of our system is to detect bad code practise and inefficient code in student solution and generate auto-feedback to guide students towards production level code.

Traditional JavaScript code interpreters are slow. As more intensive and important computation are carried out in the browser and node.js[4], the performance of JavaScript program becomes the new era of competition among all major JavaScript engine vendors. In recent years, there has been great performance improvement in JavaScript engines, and browsers like Google Chrome and Mozilla Firefox have arguably some of the fastest JavaScript engines in existence now. And many research has been conducted on improving JavaScript engine performance. Based on these research most modern browsers employ some form of compilation and optimization of JavaScript code into machine code or intermediate representation at runtime. Such kind of compilation is called Just-In-Time (*a.k.a.* JIT) Compilation. JIT compilation helps to significantly increase execution speed of JavaScript programs. Substantial research has been done to improve JavaScript engines performance on either benchmarks or real-world websites. However, not many research is conducted on the other direction: pinpointing and refactoring inefficient JavaScript code patterns to make them

[4]Node.js is a platform for server-side JavaScript.

run faster on existing JavaScript engines.

We study how major JavaScript engines works and and identify several inefficient code patterns that may slow down the program. We implemented the JIT-compiler unfriendly code detector which monitors the program execution and detect inefficient patterns at runtime.

Based on some coding guidelines[5] specified by JIT compilers and our study into those JavaScript engines, we list several rules for composing efficient code are:
(1) Initialize all object members in constructor functions (so that the instances do not change type later).
(2) Always initialize object members in the same order.
(3) Make sure that a variable has a single consistent type throughout an execution. This guideline becomes more important if the variable holds an integer value.
(4) Use contiguous keys starting at 0 for arrays.
(5) Do not load uninitialized or deleted elements.
(6) Do not store non-numeric values in numeric arrays.
(7) Monomorphic use of operations is preferred over polymorphic operations.
There are similar guidelines for other JIT compilers. Some of the guidelines such as (2) are applicable to other JIT compilers. We implement those checkers in our dynamic analysis framework to detect any of those inefficient code patterns in the student solution when testing the program.

11. EXPERIMENTAL RESULTS

This sections first briefly introduce the implementation of our tool, and followed by the description of evaluating the correctness and performance of our dynamic analysis framework by testing our framework on several well known benchmarks. The results show that after transformation, the semantics of the target program is preserved, and the slowdown caused by our framework for web page rendering and dynamic visual effect is not obvious.

11.1 Implementations

Our system is purely implemented in JavaScript and running fully in the browser to support online education. The entire system contains over 10K lines of code which implements code editing, code instrumentation, evaluation, testing generation, bug localization and program analysis. All those tasks are carried out in the browser at client side and consequently lightens the use of server which further enables massive number of students using the system simultaneously.

11.2 Benchmarks

We created our benchmark set with problems taken from the Introduction to Data Structure (CS 61B) at Berkeley as well as implementation of classic data structures and algorithms. A brief description of each benchmark program is listed as follows:

- `QSort`: Implement quick sort algorithm that takes an array of integers as input.
- `DLlist`: Implement Double Linked List data structure.
- `BTree`: Implement Binary Search Tree data structure.
- `HSort`: Implement heap sort algorithm that takes an array of integers as input.
- `MSort`: Implement merge sort algorithm that takes an array of integers as input.
- `Max`: Implement max function that returns the maximum value among the inputs.
- `Smoosh`: takes an array of integers and replaces consecutive duplicate numbers.
- `Squish`: takes an array of integers and whenever multiple consecutive items are equivalent, removes duplicate elements so that only one consecutive copy remains.

We search the Internet to find implementations written in JavaScript or if no implementation publicly available we implement them manually. To simulate the use of these programs in the real scenario, we seed bugs according to mistakes commonly made by students [40]. The bug patterns we seeded in the implementations are as follows:

Err-1	`a[i]` →	`a[i+1]`|`a[i-1]`
Err-2	`v=n` →	`v=n+1`|`v=n-1`|`v=0`
Err-3	`v1` *op* `v2` →	`v1+1` *op* `v2` | `v1-1` *op* `v2` `v1` *op* `v2+1` | `v1` *op* `v2-1`
Err-4	`return v` →	`return v+1` | `return v-1`
Err-5	`a<b` →	`a<=b` | `a>b` | `a>=b`
Err-6	`a==b` →	`a!=b`
Err-7	`a!=b` →	`a==b`
Err-8	`arrIdx` →	`arridx` | `Arridx` | ...

op ∈ { `+`, `-`, `*`, `/`, `%` }

Figure 6: Bugs seeded in benchmark programs

11.3 Bug Localization Accuracy

In this section we present an empirical evaluation that measure the accuracy of our bug localization feedbacks.

Evaluation Metric. We compare the effectiveness of our bug localization component based on diagnostic cost that calculates the percentage of statements examined to locate a fault, which is commonly used in the literature [22, 1, 26]. The diagnostic cost is defined as follows:

$$Cost = \frac{|\{j \mid f_{T_S}(d_j) \geq f_{T_S}(d_*)\}|}{|\mathcal{D}|}, \quad (1)$$

where $\mathcal{D}$ consists of all program elements appearing in the input program. We calculate the cost as the percentage of elements that developers have to examine until the root cause of the failures (d_*) are found. Since multiple program elements can be assigned with the same suspicious score, the numerator is considered as the number of program elements d_j that have larger or equal suspicious scores as that of d_*.

Evaluation Assumption. To measure the effectiveness of interactive fault localization methods with user feedback, we assume that:

1. User inspects program elements following recommended list from the most suspicious to the least.

2. Although our framework provides an option for a user to submit either no or multiple feedback. For the simplicity of evaluation, we assume that a feedback is submitted after each program element is inspected and a user will keep labeling until faults are found.

[5] `http://www.html5rocks.com/en/tutorials/speed/v8/`

Benchmark/Error	Err-1	Err-2	Err-3	Err-4	Err-5	Err-6	Err-7	Err-8	Avg(Err)
`Max`	ϕ	14.28%	ϕ	14.28%	14.28%	ϕ	ϕ	14.28%	14.28%
`QSort`	51.11%	4.44%	35.56%	35.56%	6.67%	2.22%	ϕ	2.22%	19.68%
`MSort`	2%	18%	ϕ	ϕ	28%	16%	ϕ	2%	13.2%
`HSort`	60.32%	3.17%	1.59%	ϕ	ϕ	ϕ	14.29%	1.59%	16.19%
`DLlist`	ϕ	1.37%	24.68%	ϕ	5.48%	12.33%	ϕ	8.22%	10.41%
`BTree`	ϕ	1.32%	1.32%	0.66%	1.32%	ϕ	26.49%	1.99%	5.52%
`Smoosh`	56.25%	18.75%	18.75%	ϕ	56.25%	ϕ	56.25%	12.5%	36.46%
`Squish`	56.25%	18.75%	18.75%	ϕ	56.25%	ϕ	56.25%	12.5%	36.46%
Avg(Prgm)	45.19%	10.01%	16.78%	16.83%	24.04%	10.18%	38.31%	6.91%	

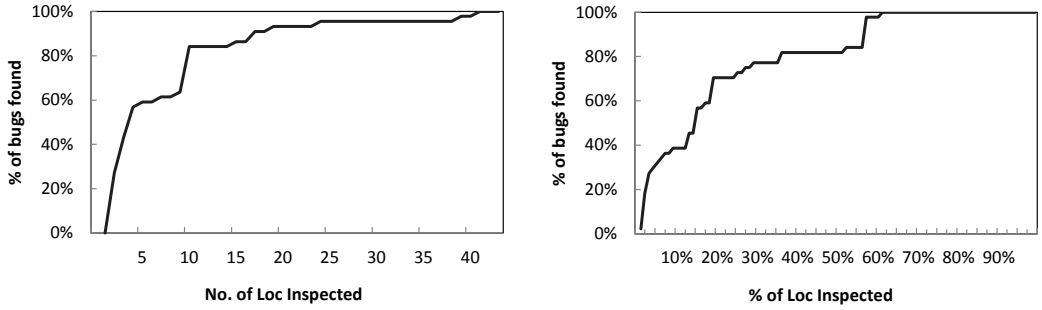

Figure 5: Bug Localization Accuracy.

Evaluation Result. Figure 5 shows all the bugs seeded in benchmark programs. ϕ means the there exists no corresponding programming constructs in the solution to seed the bug. Numbers in the table shows the percent of lines of code to inspect to identify the real bug. For example, `QSort` needs inspecting 2% of the code to find the bug seeded by pattern Err-2 in Figure 6. On the left bottom side of Figure 5, the graph shows the number of code inspected and the corresponding percent of bugs located. For instance, inspecting 5 lines of code on average can locate 60% of bugs, and inspecting 12 lines of code on average can locate over 80% of bugs.

To simulate student solution in the experiment, we search the Internet to use publicly available code snippet. Surprisingly, we find that top 5 search results of "JavaScript Quick Sort" contain unreported bugs. Those implementations are written for blogs explaining how quick sort works, but the authors only write a few test cases to show the code works without comprehensive testing. Our system detects those bugs by high coverage testing, user only need to simply click "Run". One of the bugs in Quick Sort from `Github Gist` is easily located by our system and we fix the bug and report it back to the user [6].

11.4 Conformance Test

To make sure that the program instrumentation does not change the semantics of the program. We evaluate our dynamic program analysis framework under conformance test. Test262 is a test suite intended to check agreement between JavaScript implementations and ECMA-262, the ECMAScript Language Specification (currently 5.1 Edition). The test suite contains thousands of individual tests, each of which tests some specific requirements of the ECMAScript Language Specification.

The Firefox SpiderMonkey failed 111 conformance among 1236 conformance test The instrumented SpikderMonkey failed 211 conformance tests. Due to limited time we have, we did not go through those failures one by one but an overview shows that the additional failures are caused by removing the `"use restrict;"` clause from the target code. We remove the clause because it prevents us from using some of the advanced dynamic features of JavaScript (*e.g.,* `arguments.callee`, `eval` *etc.*).

11.5 Performance Test

Our system transforms programs into intermediate representation, and thus perform program profiling and program analysis. In this section, we measure the performance of the dynamic analysis framework on both SunSpider benchmark and our assignment benchmarks.

Testing on the programming assignment benchmarks we collected show that running the entire process(see Figure 2) takes 2s at most on each program.

SunSpider is a benchmark suite that aims to measure JavaScript performance on tasks relevant to the current and near future use of JavaScript in the real world, such as encryption and text manipulation. The suite further attempts to be balanced and statistically sound. The result of SunSpider Benchmarks on our platform is shown in Table 2. On average, the code after transformation is 150X slower than the original target code. The slowdown drastically increases on this benchmark because most of the calculation is performed using the JavaScript code which will be fully transformed. But for most real-world websites, the JavaScript code invokes lower level native library functions provided by the browser (implemented in C/C++) to perform most compute-intensive tasks. And since our framework is only interested in and transforms the top level JavaScript code, the slowdown in this scenario is much less. Although the slowdown on SunSpider seems intimidating, evaluation on the programming assignment benchmarks shows that the entire feedback generating process (including testing, bug

[6] `https://gist.github.com/springuper/4182296`

Table 2: SunSpider Benchmark Running on Firefox Before and After Transformation.

TEST	COMPARISON	TIME(TRANSFORMED)	TIME(ORIGINAL)
3d:	140.6x	4582.8ms +/- 2.30%	32.6ms +/- 5.80%
cube:	95.2x	1294.6ms +/- 2.60%	13.6ms +/- 13.90%
morph:	291.7x	1575.2ms +/- 3.80%	5.4ms +/- 12.60%
raytrace:	126.0x	1713.0ms +/- 5.70%	13.6ms +/- 5.00%
access:	333.0x	6060.8ms +/- 2.50%	18.2ms +/- 17.00%
binary-trees:	612.1x	1713.8ms +/- 4.30%	2.8ms +/- 19.90%
fannkuch:	391.2x	2582.0ms +/- 1.60%	6.6ms +/- 10.30%
nbody:	257.7x	1185.4ms +/- 9.90%	4.6ms +/- 52.70%
nsieve:	138.0x	579.6ms +/- 3.70%	4.2ms +/- 13.20%
bitops:	576.0x	5183.6ms +/- 1.70%	9.0ms +/- 25.80%
3bit-bits-in-byte:	1814.0x	1451.2ms +/- 4.60%	0.8ms +/- 69.50%
bits-in-byte:	654.1x	2093.0ms +/- 2.10%	3.2ms +/- 32.50%
bitwise-and:	357.6x	643.6ms +/- 3.70%	1.8ms +/- 30.90%
nsieve-bits:	311.2x	995.8ms +/- 1.10%	3.2ms +/- 17.40%
controlflow:	957.2x	2488.6ms +/- 1.50%	2.6ms +/- 26.20%
recursive:	957.2x	2488.6ms +/- 1.50%	2.6ms +/- 26.20%
crypto:	215.2x	4089.2ms +/- 3.40%	19.0ms +/- 15.30%
aes:	134.0x	1340.4ms +/- 4.10%	10.0ms +/- 21.50%
md5:	289.7x	1332.4ms +/- 4.70%	4.6ms +/- 14.80%
sha1:	321.9x	1416.4ms +/- 5.10%	4.4ms +/- 15.50%
date:	91.3x	2336.4ms +/- 4.10%	25.6ms +/- 12.70%
format-tofte:	127.3x	1730.8ms +/- 3.00%	13.6ms +/- 10.40%
format-xparb:	50.5x	605.6ms +/- 13.50%	12.0ms +/- 19.40%
math:	229.5x	4315.0ms +/- 3.90%	18.8ms +/- 5.50%
cordic:	777.8x	1866.6ms +/- 3.50%	2.4ms +/- 28.40%
partial-sums:	60.2x	866.4ms +/- 4.10%	14.4ms +/- 4.70%
spectral-norm:	791.0x	1582.0ms +/- 8.10%	2.0ms +/- 0.00%
regexp:	1.13x	23.4ms +/- 2.90%	20.8ms +/- 2.70%
dna:	1.13x	23.4ms +/- 2.90%	20.8ms +/- 2.70%
string:	66.3x	5000.2ms +/- 2.80%	75.4ms +/- 22.20%
base64:	80.0x	735.8ms +/- 4.10%	9.2ms +/- 6.00%
fasta:	175.3x	1367.0ms +/- 3.60%	7.8ms +/- 17.50%
tagcloud:	50.1x	861.0ms +/- 6.40%	17.2ms +/- 27.70%
unpack-code:	36.8x	772.8ms +/- 11.90%	21.0ms +/- 16.70%
validate-input:	62.6x	1263.6ms +/- 3.90%	20.2ms +/- 68.20%
TOTAL	**153.5x**	**34080.0ms +/- 1.70%**	**222.0ms +/- 5.90%**

localization and program analysis) takes at most 2s for each assignment as the programming assignments are simple algorithms and data structures and thus do not have scalability issues.

12. RELATED WORK

`FindBugs`[21] is a static code checker that detects error-prone code patterns. In comparison, our analysis framework allows user defined analysis module to detect dynamic operation patterns. Our dynamic analysis framework also allows building other checkers that can help improve static checkers. For example, `JSLint` is a JavaScript static code checker that detects the evil use of JavaScript programming language[10]. In which it deprecates the use of function `eval` considering performance issues and the unpredictable behaviour. `JSLint` can only detect the explicit use of `eval` due to the limitation of alias analysis, and the detection might be unsound as the call expression might never be executed. Our analysis framework on the other hand is capable of accurately detecting both explicit and implicit use of `eval` function and thus can be used as a substitution or supplement to those static checkers.

Many static and dynamic analysis tools [33, 43, 4, 18] for JavaScript have been proposed. Richards *et. al.* did an empirical study[35] on the use of dynamic behaviours of JavaScript programming language in real-world websites. Their results show that most type-checking techniques for JavaScript are based on some strong assumption which are rarely hold in real-world web applications. Newsome *et. al.* implemented a dynamic taint analysis framework[31] for security analysis in web browser. Feldthaus *et. al.* proposed an automatic refactoring tool[17] to avoid using the evil parts of JavaScript code.

Our JavaScript dynamic analysis framework is build on top of `Jalangi`[36, 37] which is a dynamic analysis framework mainly for back-end JavaScript on `Node.js`. But Vast majority of JavaScript code is written for front-end running in web browsers and before `Jalangi` and `Jalangi` does not fully support front-end JavaScript engine like Firefox. Before our framework, there exists no dynamic analysis framework for front-end in browser analysis for JavaScript similar to `Valgrind`[30], `PIN`[27] or `DynamoRIO`[6] for x86. So we design and implement this in-browser framework for front-end JavaScript dynamic analysis. Based on `Jalangi`, we refactored and rewrote some critical components of the original framework to make it fully support and easily analysing any real-world web pages. Further more, our front-end analysis framework completely decoupled from the JVM and can seamlessly work with web console and debugger in the browser which reuses the UI provided by Firefox. This facilitates developers more intuitive interaction such as analysis module plug in and remove on the fly, and thus makes our framework even more powerful.

13. CONCLUSION

In this paper we present a system for automatically providing feedback for introductory programming assignments for JavaScript. The system is a test-based approach that can complement formal-method-based approaches and provides more comprehensive testing and more utilities against existing test-based auto-grading systems. The technique uses symbolic execution, statistical bug localization, program analysis to automatically grade, recommend bug locations, and generate program analysis report to students. We have evaluated our technique on a set of benchmarks we collect and the results show that inspecting 5 lines of code recommended by our system, 60% of bugs common made by students can be located. We believe this technique can be a useful platform for not only massive online education courses, but also code interviewing and JavaScript development.

A demo of the Auto-grading system is available at:

`https://www.eecs.berkeley.edu/~gongliang13/jalangi_ff/fault_loc.htm`

14. REFERENCES

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 2009.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *TAICPART-MUTATION*, 2007.

[3] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of dfa constructions. In F. Rossi, editor, *IJCAI*. IJCAI/AAAI, 2013.

[4] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In A. P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 428–452. Springer, 2005.

[5] H.-J. Boehm and C. Flanagan, editors. *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM, 2013.

[6] D. Bruening, T. Garnett, and S. P. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO*, pages 265–275. IEEE Computer Society, 2003.

[7] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In R. Draves and R. van Renesse, editors, *OSDI*, pages 209–224. USENIX Association, 2008.

[8] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[9] Codility. *https://codility.com/train/*.

[10] D. Crockford. *JavaScript - the good parts: unearthing the excellence in JavaScript*. O'Reilly, 2008.

[11] L. M. de Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

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

[13] S. ECMA-262". *http://www.ecma-international.org/publications/standards/Ecma-262.htm*.

[14] G. V. Engine. *http://code.google.com/p/v8/*.

[15] N. J. Engine. *http://www.webkit.org/projects/javascript/index.html*.

[16] N. S. S. J. Engine. *http://nodejs.org/*.

[17] A. Feldthaus, T. D. Millstein, A. Møller, M. Schäfer, and F. Tip. Refactoring towards the good parts of javascript. In *OOPSLA Companion*, 2011.

[18] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for javascript ide services. In *ICSE*, 2013.

[19] P. Godefroid. Test generation using symbolic execution. In D. D'Souza, T. Kavitha, and J. Radhakrishnan, editors, *FSTTCS*, volume 18 of *LIPIcs*, pages 24–33. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[20] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.

[21] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[22] J. Jones and M. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, 2005.

[23] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In D. F. Redmiles, T. Ellman, and A. Zisman, editors, *ASE*, pages 273–282. ACM, 2005.

[24] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[25] C. Lattner and V. S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004.

[26] Lucia, D. Lo, L. Jiang, and A. Budi. Comprehensive evaluation of association measures for fault localization. In *ICSM*, 2010.

[27] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[28] B. Meyer, L. Baresi, and M. Mezini, editors. *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. ACM, 2013.

[29] P. A. Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound boolean predicates. In *ISSTA*, pages 5–15, 2007.

[30] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[31] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

[32] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, pages 199–209, 2011.

[33] B. Petrov, M. T. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *PLDI*, 2012.

[34] M. Rhino. *https://developer.mozilla.org/en-US/docs/Rhino*.

[35] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *PLDI*, 2010.

[36] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In Meyer et al. [28], pages 488–498.

[37] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of javascript. In Meyer et al. [28], pages 615–618.

[38] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In M. Wermelinger and H. Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005.

[39] C. H. Shuvendu Lahiri, Rohit Sinha. Automatic rootcausing for program equivalence failures. *MSR Tech Report*, 2014.

[40] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In Boehm and Flanagan [5], pages 15–26.

[41] M. SpiderMonkey. *https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey*.

[42] M. TraceMonkey. *https://wiki.mozilla.org/JavaScript:TraceMonkey*.

[43] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *POPL*, 2007.