# A Practical Blended Analysis for Dynamic Features in JavaScript

Shiyi Wei and Barbara G. Ryder
Department of Computer Science
Virginia Tech
{wei, ryder}@cs.vt.edu

***Abstract*—JavaScript is widely used in Web applications; however, its dynamism renders static analysis ineffective. Our JavaScript Blended Analysis Framework is designed to handle JavaScript dynamic features. It performs a flexible combined static/dynamic analysis. The blended analysis focuses static analysis on a dynamic calling structure collected at runtime in a lightweight manner, and refines the static analysis using dynamic information. The framework is instantiated for points-to analysis with stmt-level MOD analysis and tainted input analysis. Using JavaScript codes from actual webpages as benchmarks, we show that blended points-to analysis for JavaScript obtains good coverage (86.6% on average per website) of the pure static analysis solution and finds additional points-to pairs (7.0% on average per website) contributed by dynamically generated/loaded code. Blended tainted input analysis reports all 6 true positives reported by static analysis, but without false alarms, and finds three additional true positives.**

***Index Terms*—JavaScript, program analysis, points-to analysis, taint analysis**

## I. Introduction

In the age of SOA and cloud computing, JavaScript has become the *lingua franca* of client-side applications. Web browsers act as virtual machines for JavaScript programs that provide flexible functionality through their dynamic features. Recently, it was reported that 98 out of 100 of the most popular websites (*http://www.alexa.com*) use JavaScript [9]. Many mobile devices – smart phones and tablets – use JavaScript to provide platform-independent functionalities. Unfortunately, the dynamism and flexibility of JavaScript is a double-edged sword. The dynamic constructs enable programmers to easily create client-side functionalities at the cost of rendering static analysis ineffective in their presence. However, these constructs often provide opportunities for security exploits.

Given the ubiquity of JavaScript, it is important for researchers to address possible problems in security, code optimization, performance diagnosis, debugging, etc. Several analysis approaches have been proposed to detect/prevent security vulnerabilities in JavaScript applications, such as cross-site scripting [2], [17], and to improve performance through trace-based, just-in-time compilation techniques [6]. These methods use either static or dynamic analysis or a combination of both. Nevertheless, the reflective features of JavaScript thwart the effectiveness of static analysis in addressing these problems, and dynamic analysis either covers too few possible runtime situations or is too costly in terms of overhead. All these approaches have left great room for improvement in the handling of the dynamism of JavaScript in real-world applications.

Recent studies [18], [21], [20] reveal that JavaScript programs are full of dynamic features, and that the dynamic behavior of actual websites confirm this fact. There are several mechanisms in JavaScript whereby executable code can be generated at runtime, (e.g., $eval$). Richards $et.al$ [20] show that $eval$ and its related language structures are widely used in real Web applications. In JavaScript programs, a function can be called without respecting the declared number of arguments; that is, functions may have any degree of variadicity so that it is hard to model them statically. In Richards $et.al$ [21], variadic functions are shown to be common and the occurrence of functions of high variadicity is confirmed. JavaScript call sites and constructors are quite polymorphic. These dynamic features make it hard to precisely reason about JavaScript applications; existing work tends either to ignore or to make incorrect assumptions regarding them [21].

Dealing with dynamic language constructs has been addressed previously in analyses of Java. For example, in our own work focused on performance diagnosis of framework-intensive Java programs, we dynamically collected a problematic execution and performed a static escape analysis on its calling structure [4], [5] (see section V). Java features such as reflective calls and dynamically loaded classes were recorded by the dynamic analysis, allowing more precise modeling than by pure static analysis (i.e., an analysis based on monotone data-flow frameworks [16]).

Our JavaScript blended analysis captures richer information about dynamic language features, including dynamically generated/loaded JavaScript code (e.g., through $eval$s or interpreted *urls*) and variadic function usage. Because of the widespread usage of these dynamic features in JavaScript applications, their capture is important because a pure static analysis may miss them (e.g., when an $eval$ contains a JavaScript code string which contains user inputs) or approximate them in the worst case (e.g., treating all variadic functions with the same signature as the same function because they cannot be differentiated at compile time).

We have designed a new general-purpose, JavaScript Blended Analysis Framework that facilitates analysis of the dynamic features in JavaScript. Our framework is an investigation of how to design a practical analysis for a general-purpose scripting language while accommodating the dynamic features

of the language. Our goal is to judiciously combine dynamic and static analyses in an unsafe [16] analysis of JavaScript, to account for the effects of dynamic features not seen by pure static analysis, while retaining sufficient accuracy to be useful. In our framework, we analyze multiple executions of a JavaScript code in order to obtain analysis results for the entire program.

We have instantiated our JavaScript Blended Analysis Framework to do *points-to analysis with stmt-level MOD* as well as *tainted input analysis*, in order to show time cost and precision differences in empirical comparison with corresponding pure static analyses. Points-to analysis of JavaScript is an important *enabling* analysis, supporting many static clients. Our blended points-to analysis is used by stmt-level MOD to ascertain which objects may experience side effects at property assignments. Our tainted input analysis tracks the propagation of user input data to detect possible integrity violations in JavaScript programs.

The major contributions of this paper are:

**JavaScript Blended Analysis Framework.** Our flexible framework is designed to better analyze the language's dynamism and to allow replacement of components in different contexts (e.g., choices of a specific dataflow analysis). The framework is implemented using *TracingSafari* [21], and the open-source *IBM T.J. Watson Libraries for Analysis* (*WALA*).[1]

**Implementation of blended client analyses for JavaScript.** The framework is instantiated for two clients, points-to analysis with stmt-level MOD and tainted input analysis, on JavaScript codes from popular websites. By obtaining the dynamically generated code online as well as call target and object creation information, we are able to analyze several dynamic features of JavaScript including code within an $eval$, function variadicity, and dynamic type-dependent object creation.

**Experimental evaluation.** The empirical results of blended and pure static analyses are compared on the two implemented client problems. The comparison shows that (i) blended analysis for JavaScript obtained good coverage of the static analysis points-to solution (i.e., 78.0%-93.0% of static analysis results on average per website) and (ii) found additional (i.e., missed) points-to pairs resulting from dynamically generated/loaded code. The latter comprised 1.4%-9.9% of the final solution on average per website, quite significant for some websites. The cost of the blended analysis was shown to exceed the cost of static analysis by only 31.6% on average per website. The stmt-level MOD client results indicate that blended analysis produces 46.7% fewer objects to look at while debugging and provides additional objects to examine in dynamically generated/loaded code. Blended tainted input analysis reported all 6 true positives found by pure static analysis, but without any false alarms; blended analysis also managed to find three additional true positives in dynamically generated/loaded code.

**Overview.** The rest of this paper is organized as follows: section II uses an example to illustrate the dynamism of JavaScript. Section III introduces the JavaScript Blended Analysis Framework and the two client problems. Section IV presents our experimental results, section V discusses related work, and section VI offers conclusions and future work.

[1]http://wala.sourceforge.net/

```
1  function blog(input){
2
3    if(arguments.length == 1) {
4      var v  = new Blog(input);
5      var process = newblog;
6    }else if (arguments.length == 2) {
7      var i = eval(arguments[1]);
8      var v = new Comment(input, i);
9      var process = updateblog;
10   }
11
12   p = process(v);
13   document.write(p);
14 }
```

**Fig. 1:** JavaScript example.

## II. MOTIVATING EXAMPLE

In Figure 1 we present a sample JavaScript program containing dynamic characteristics to illustrate the challenges of analyzing dynamic languages. The example illustrates a JavaScript function *blog*, which takes the *input* and either creates a new blog or adds a comment to an existing blog.

In line 1, the function signature is given with one argument; however, it is designed to be called with different arguments. The code can behave with totally different functionality, depending on the number of arguments. Lines 3 to 5 execute if $arguments.length$ is 1. Lines 6 to 9 demonstrate another case when 2 arguments are provided. In these two branches, different objects ($Blog$ in line 4 and $Comment$ in line 8) can be initialized and the variable $process$ is assigned different values. Line 7 uses $eval$ to read the string associated with the value of $arguments[1]$ to provide the index of the existing blog. Whether or not this code is executed properly can be decided only at runtime since statically we cannot learn the value of $arguments[1]$. The code actually executed in line 12 depends on the value of $process$, ($newblog(v)$ when $arguments.length = 1$ or $updateblog(v)$ when $arguments.length = 2$). Line 13 writes the returned page into the document.

This example can be exploited for security vulnerability if the server does not properly sanitize [9] the user inputs. For example, an user input containing executable JavaScript code can cause cross-site scripting when it is written into the document. Program analysis is needed to detect the vulnerabilities; however, static analysis cannot precisely model this code because of the dynamic features. Our blended analysis models the JavaScript features listed in Table I with the dynamic information collected.

## III. BLENDED ANALYSIS OF JAVASCRIPT

In this section we present an overview of our JavaScript Blended Analysis Framework shown in Figure 2.

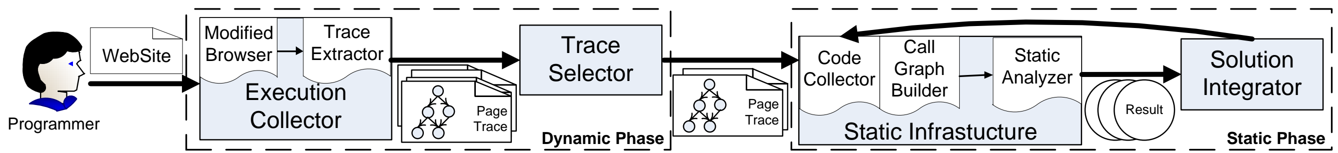


**Fig. 2:** JavaScript Blended Analysis Framework

**TABLE I:** JavaScript features handled by blended analysis

| Feature | Instance |
|---|---|
| dynamic code loading | src = "www.sample.com/"+name+".js" |
| function variadicity | function $blog(input)$ in Fig. 1 |
| $eval$ construct | $eval$ in Line 7, Fig. 1 |
| type-based object creation | variable $v$ in Lines 4 and 8, Fig. 1 |
| type-based dynamic dispatch | call $process(v)$ in Line 9, Fig. 1 |

### A. Framework

The JavaScript Blended Analysis Framework was designed to be a practical, general-purpose combination of dynamic and static analyses capable of capturing the effects of the dynamic features of JavaScript, especially those that lead to run-time code generation. Specifically, the framework aims to offer an efficient methodology to obtain a better analysis solution than a pure static analysis of a scripting language with dynamic types and many late-binding constructs.

Our focus is on JavaScript code found on webpages; we refer to code on a single webpage as a JavaScript *program*. Our framework is targeted to analyze such programs, which have been shown to have attributes different from some standard JavaScript benchmarks such as *SunSpider*[2] and *V8*[3] [21], [18].

Figure 2 illustrates the components in our framework. In the *Dynamic Phase*, the *Execution Collector* gathers run-time information about a JavaScript program and the *Trace Selector* selects a subset of the observed executions which offer good (method) coverage of a program. The goal is to obtain good program coverage at lower cost than that of using all the executions. In the *Static Phase*, the *Code Collector* finds the JavaScript code that was executed, including both statically visible and dynamically generated/loaded code. The *Call Graph Builder* creates a call graph from the calls recorded and stores other collected method-specific information as node annotations. Because our blended analysis aims to handle the dynamism of JavaScript, the call graph may include dynamic information that would be very hard (or impossible) for a pure static analysis to approximate well. The *Static Analyzer* analyzes the program as represented by the call graph. The *Solution Integrator* combines dataflow solutions from different executions into a program solution, and decides if there are more executions to analyze.

The JavaScript Blended Analysis Framework is a workflow in which individual components are replaceable. For example, by varying the instrumentation applied to the browser, we can collect different sorts of information. By changing the *Static Analyzer*, we can change the specific analysis applied to the JavaScript program as we did in our experiments.

An overall goal of our research is to ascertain how well our JavaScript Blended Analysis Framework can analyze programs written in a heavily dynamic, late binding language. We also want to better understand the tradeoffs between results obtained by a conservative, pure static analysis versus those from a practical, unsafe blended analysis.

### B. Dynamic Phase

Our *Execution Collector* relies on a specialized version of *TracingSafari*, an instrumented version of WebKit[4] JavaScript engine developed for characterizing the dynamic behavior of JavaScript programs [21]. This tool records operations performed by the JavaScript interpreter in Safari including *reads, writes, field deletes, field adds, calls*, etc. It also collects events such as garbage collection and source file loads. Since our dynamic analysis infrastructure needs to be lightweight, we modified *TracingSafari* to collect only the information required by blended analysis, namely, method call information and object instance creation including types.

By *lightweight*, we mean an analysis that developers will be willing to use to obtain high-quality analysis results, even though browser performance may degrade somewhat. In our experiments (see section IV), we ran both our modified instrumented Safari and an original Safari on *JSBench* [19], a JavaScript benchmark generated from real websites, to observe the instrumentation overhead. The results showed that the instrumented Safari took on average 127.86ms, while the original Safari finished on average in 89.57ms. Thus in this fairly realistic trial, our instrumented Safari ran 42.7% slower than the original. Since we are using a research prototype not optimized for performance, we believe this factor can be greatly reduced.

An execution of a website may involve code on several different webpages. The sequence of JavaScript instructions collected during an execution is decomposed into *page traces*; each trace is a consecutive sequence of instructions from the same webpage. There is at least one trace generated for each page executed containing JavaScript code. A webpage (i.e., a JavaScript program) usually is analyzed on the basis of several traces. A page trace consists of a dynamic call tree, recorded object creations, compile-time visible JavaScript source code and dynamically generated/loaded code including any executed library code. The *Trace Extractor* builds the set of page traces corresponding to each webpage from a set of

[2]http://www.webkit.org/perf/sunspider/sunspider.html
[3]http://v8.googlecode.com/svn/data/benchmarks/v7/run.html
[4]webkit.org

recorded website executions. In Figure 2, an analysis solution is computed for a JavaScript program, that is, a set of page traces from the same webpage associated with these recorded executions.

Because JavaScript is a dynamically typed programming language, static reasoning about JavaScript types may not be very precise in situations where actual object types are determined by run-time assignments, for example. Pure static analysis approximates object types which can introduce imprecision in situations such as lines 4,8 in Figure 1. In blended analysis, our *Execution Collector* records the exact types of the objects created within each method in the call graph. With this additional information, blended analysis can more precisely model executions, both inter-procedurally and intra-procedurally.

Pure static analysis can extract JavaScript source code from webpages; however at runtime, invocations of reflective constructs such as $eval$ may generate new JavaScript code. This generated code may be difficult to model statically because of $eval$ parameters containing values set at runtime. In addition, webpages often download JavaScript codes and load them dynamically. Pure static analysis may have no access to this downloaded code whereas our *Execution Collector* is able to capture all the source code file loads, capturing all code generated or loaded. The behavior of this dynamically generated/loaded code is captured the same as the other parts of the program.

Function variadicity occurs when a function can be called with an arbitrary number of arguments, regardless of its declaration. If fewer arguments are provided than in the declaration, the values of the rest of the declared arguments are set to be $undefined$. If more arguments are provided than in the declaration, the arguments can be accessed through an associated $arguments$ variable. Sometimes, the actual behavior within a function can be differentiated by its number of arguments as in the *blog* function of Figure 1. Existing pure static analyses for JavaScript normally ignore this feature because the actual arguments provided during the call can only be known at runtime. In contrast, our *Execution Collector* captures the actual number of arguments for each call so that the *Call Graph Builder* can build separate nodes in the call graph for instances of the same signature function with different numbers of arguments, introducing some context sensitivity into the call graph.

**Selecting traces.** Our JavaScript Blended Analysis Framework uses multiple executions, because we would like to explore as much of the program as possible for some problems; however, those executions may overlap in the methods they cover. The *Trace Selector* tries to minimize the number of traces analyzed, while covering as much program behavior as possible. We hypothesize that there is a subset of the traces that can be used as the program representation without much loss of precision in the analysis solution. Our goal is to achieve a static analyses result on the subset of traces that is the same as or *close to* the solution obtained using *all* traces, while reducing the overall cost of blended analysis.

**Input:** $Tr$(All traces from one webpage collected by dynamic analysis); $Th$(Threshold of the selection algorithm)
**Output:** $Tr_{best}$(Traces selected to do static analysis)

1: $tr_i \leftarrow random(Tr)$
2: $Dist \leftarrow Th$
3: **while** $Dist \geq Th$ and $Tr$ is not empty **do**
4: $\quad Tr - \{tr_i\}$
5: $\quad Tr_{best} \leftarrow Tr_{best} \bigcup \{tr_i\}$
6: $\quad Dist \leftarrow -1$
7: $\quad$ **for** each $t$ in $Tr$ **do**
8: $\quad\quad$ **if** $dist(Tr_{best}, t) > Dist$ **then**
9: $\quad\quad\quad Dist \leftarrow dist(Tr_{best}, t)$
10: $\quad\quad\quad tr_i \leftarrow t$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: **end while**

**Fig. 3:** *Trace Selector* algorithm

The trace selection algorithm in Figure 3 works as follows. The set of all the collected traces from the same webpage is input to this selection algorithm. At line 1, a trace $tr_i$ is randomly chosen to be the first selected from all the traces. Line 2 initializes the variable $Dist$ used as a criterion to select traces. During the algorithm, $Dist$ is a heuristic score of what a particular trace will add to those traces already in the selected set $Tr_{best}$. Lines 3-13 describe the selection process which continues until the threshold $Th$ is reached or there are no longer any traces to add (i.e., line 3). The threshold $Th$, a value between 0 and 1, is an input to the algorithm that can be adjusted by the user of the framework. Lines 4 and 5 process the sets by removing the selected trace from $Tr$ and adding it to $Tr_{best}$. Lines 7 to 12 comprise a loop to select the best next candidate trace to add. The $dist$ function compares a candidate trace with those in the already selected set $Tr_{best}$ to calculate the corresponding $Dist$ value. The trace with the largest $Dist$ value is chosen as the next candidate to add. The set of traces selected by *Trace Selector* is an input to the *Static Phase*.

Our current heuristic, used by the $dist$ function in line 8 of Figure 3, is comprised of three factors: method coverage (i.e., $dist_{func}$), created object type coverage (i.e., $dist_{obj}$), and dynamically generated/loaded code coverage (i.e., $dist_{dyn}$). It calculates a weighted linear combination of these factors to emphasize using traces that cover more methods, explore more different object types and contain as much as possible of the dynamically generated/loaded code encountered. Covering more methods explores more code; covering more observed object types, in the presence of dynamic dispatch, potentially explores more program paths. The hypothesis is that these two heuristics will enable blended analysis to find the same (true) solution elements that would be found by a pure static analysis. Covering the dynamically generated/loaded code will enable blended analysis to add to a solution found by a pure static analysis which has no access to the corresponding code. The heuristic combination of factors with a chosen threshold maintains the balance between blended analysis performance (i.e., fewer traces) and accuracy (i.e., more traces).

Our heuristic combines all three factors each of which is normalized to fall between 0 and 1. In our current implementation, the $dist(TT, t)$ function returns the value of: $0.5 \times dist_{dyn} + 0.4 \times dist_{func} + 0.1 \times dist_{obj}$, where $dist_x$ represents the x factor calculated when trace $t$ is compared to the set of traces $TT$.[5] Although this was our initial choice of weights, it seems to have worked well for us. In our future work, we will explore the sensitivity of this heuristic to changes in these weights as well as changes in the threshold adopted for different analyses and for specific analysis clients. Note that this heuristic is not specific to JavaScript but can be used for other dynamic languages with similar characteristics.

### C. Static Phase

We built our static infrastructure for analyzing JavaScript on the *IBM T.J. Watson Libraries for Analysis (WALA)* open-source framework, a static analysis framework for Java that includes a JavaScript front-end. *WALA* parses JavaScript source code from a webpage producing an abstract syntax tree (AST) and translates the AST into the *WALA* intermediate form. Several challenges of analyzing JavaScript, including prototype-chain property lookups and reflective property accesses, are addressed in *WALA* [9].

The *Call Graph Builder* builds the call graph of each page trace as a *WALA* data structure with pruned source code for each node. Since the source code of some executed, dynamically generated/loaded functions is not available to *WALA*, we implemented the *Code Collector* to obtain this code from the page trace. In addition, in *WALA* JavaScript functions are identified through source code declarations so that variadic functions cannot be distinguished statically. In our implementation, a *WALA* call graph node is extended to include a *context*, the number of arguments (i.e., *arguments.length*). Therefore, variadic functions seem to have duplicate nodes in our *WALA* call graph, but each node context is different.

Pruning is an optimization technique applied in our blended analysis for Java to each executed method's control flow graph. It was very effective in removing approximately 30% of the statements from Java functions [5]. Essentially, we can remove provably unexecuted statements in functions, by noticing which function calls and object creation sites were not recorded in the trace and using control dependence information. The *Call Graph Builder* applies this pruning technique and also uses specialized pruning on variadic functions. When branches of a variadic function are determined by the value of $arguments.length$ (see example in Figure 1), that value can be used to prune the statements on unexecuted branches to provide a more accurate approximation of the code in the function variant. A future goal of our research is to study the effect of pruning on analysis of JavaScript, especially to see if analysis precision is increased for such variadic functions.

The *Static Analyzer* of our JavaScript Blended Analysis Framework applies an inter-procedural pure static analysis to each page trace, taking input from the *Code Collector* and *Call Graph Builder*, and producing a solution. The *Solution Integrator* combines the results from multiple traces into the final blended analysis solution for the JavaScript program.

[5]More information about this heuristic can be found in [22].

**Analysis clients.** The experiments described in section IV used two clients of blended analysis: points-to analysis with stmt-level MOD and tainted input analysis, a type of information flow. The former client uses a blended points-to analysis to find targets of object property assignments. The latter client analyzes whether data can flow from an user input statement (i.e., source) to a specific sensitive statement (i.e., sink).

**Client I.** *WALA* provides an Andersen-style flow- and context-insensitive points-to analysis for JavaScript which performs on-the-fly call graph construction; we refer to this as *pure static points-to analysis* in our experiments in section IV. In order to implement *blended points-to analysis*, we modified the *WALA* implementation by substituting use of our dynamic call graph for the on-the-fly construction. *stmt-level MOD* examines every JavaScript statement that writes a value to an object property and counts the number of objects which may experience a side effect according to the points-to solution. Using stmt-level MOD results to compare two different points-to solutions, we can contrast their utility for debugging on a set of failure runs.

**Client II.** Tainted input analysis detects instruction source-sink pairs in a JavaScript program that violate program integrity, that is, that fail to prevent a tainted input from reaching a sensitive operation.

For JavaScript, we define the *sources* and *sinks* as follows: (i) a data source is *tainted* when the user has control of its value. In our tainted input analysis, JavaScript event handlers that take user inputs as parameters are considered as *sources*; the user input arguments are marked as tainted; (ii) Any object that can affect the behavior of websites or browsers is considered to be *sensitive*. In tainted input analysis, we consider *document*, *history*, *location*, and *window* as sensitive objects. The statements that call the built-in methods of these objects are *sinks* (e.g., *document.write(exp1, exp2, ...)*, *location.assign(URL)*, and *window.setTimeout(code, millisec, lang)*). Our tainted input analysis detects and reports a source-sink pair if there exists at least one flow from a tainted input into a sink statement, ignoring any possible sanitizers in the code.

Our two phase algorithm for tainted input analysis uses a call graph of the program and the JavaScript program code as inputs. The first phase performs a call graph reachability analysis to filter out any node that is not on a call path from a source to a sink; the remaining nodes are called *candidates*. The second phase performs an inter-procedural traversal of the call graph from each source through candidate nodes to any reachable sink. An intra-procedural data dependence analysis is performed on each encountered candidate method to track the tainted parameters into candidate calls. The calls of non-candidate methods are approximated: if one argument of the call is tainted, we assume all the arguments are tainted because we do not analyze the code. Call cycles are handled by fixed point iteration.

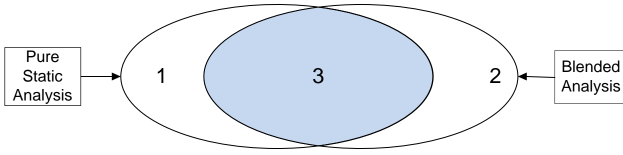


**Fig. 4:** Relation between solutions obtained through pure static analysis vs. blended analysis

### *D. Blended vs Pure Static Analysis*

The diagram in Figure 4 reflects the relationship between a pure static analysis and a blended analysis for JavaScript applied to the same program. Part 1 shows that there may be a set of results reported by pure static analysis that blended analysis does not calculate because it does not explore every executable path in the program. Some of these results may be false positives introduced by over-approximation that may be avoided by the more precise call graph and/or pruning of blended analysis. Part 2 shows that the blended analysis solution may contain results missed by a pure static analysis because of difficult to model dynamic constructs in JavaScript. Part 3 of the diagram shows results reported by both analyses.

## IV. Evaluation

### *A. Experiment Design*

We conducted experiments on two analysis clients in order to observe the performance and accuracy of blended analysis for JavaScript. Points-to analysis with stmt-level MOD solves for statement-level side effects, a program characteristic broadly spread throughout the code, so that an analysis must examine as much of the code as possible. Tainted input analysis solves for a sparser program property that tracks tainted inputs to sensitive statements; examination of all the code may not be required for this client. These clients allowed us to contrast blended with pure static analysis in different analysis contexts in which we explored the following hypotheses:

**Hypothesis 1:** Blended points-to analysis for JavaScript can achieve good coverage of the true positives in a pure static solution while producing extra points-to pairs due to dynamically generated/loaded code (section IV-B).

**Hypothesis 2:** Blended stmt-level MOD analysis will help programmers in debugging more than pure static analysis, by presenting them with fewer side-effected objects to examine, while conservatively calculating those side effects which can occur on a set of failing executions (section IV-B).

**Hypothesis 3:** Blended tainted input analysis is more precise (i.e., fewer false alarms) and accurate (i.e., more true positives) than a pure static tainted input analysis (Section IV-C). This hypothesis may be generalizable to all information flow problems.

**Benchmarks.** The experiments were conducted on code from 26 websites. We chose 18 of them from the most popular websites on *www.alexa.com* and the rest because they were used as benchmarks in [13]. Dynamic statistics for these benchmarks are given in Table II. Our blended points-to analysis with stmt-level MOD was tested on the eight websites in Column 1 of Table II. Tainted input analysis was calculated using all 26 web sites because we needed more JavaScript programs to find occurences of this sparser program property.

In Table II, *page count* is the number of webpages executed at each website and then analyzed. *Trace count* is the total number of traces collected for each website. For each of the websites listed in column 6, we explored 30 traces. For each of the websites listed in column 8, we explored 10 webpages. Recall that the number of page traces chosen by *Trace Selector* determines the number of separate static analyses required by our blended analysis for a JavaScript program (see section III). The selection threshold was 0.02 in all the experiments. Columns 4 and 5 show the number of pages on which we observed $eval$ calls and variadic functions on the eight websites used for both clients.

One of the authors manually performed instrumented explorations of these websites in order to generate input for the analyses. At the time, he had no knowledge of the underlying JavaScript code at these websites.

The experimental results were obtained on a 2.53 GHz Intel Core 2 Duo machine with 4 GB memory running the Mac OS X 10.5 operating system.

### *B. Client I: Points-to Analysis with stmt-level MOD*

**Results comparison.** As shown in Table II, blended points-to analysis for JavaScript collected 4695 traces in total for the eight websites in Column 1. The *Trace Selector* chose 3677 traces to be analyzed in the static phase, eliminating 21.7% of the traces.

**TABLE III:** blended points-to analysis results

| Website | Blended coverage of static results(%) | Additional blended results(%) |
|---|---|---|
| google.com | 89.7 | 5.9 |
| facebook.com | 85.3 | 7.5 |
| youtube.com | 89.1 | 9.9 |
| yahoo.com | 78.0 | 9.8 |
| baidu.com | 93.0 | 6.7 |
| wikipedia.org | 92.1 | - |
| live.com | 81.8 | 7.5 |
| blogger.com | 83.8 | 1.4 |
| geom. mean | **86.6** | **7.0** |

Table III shows how well our blended points-to solution compares to a baseline static points-to analysis of all the JavaScript code on a webpage, averaged across the pages explored at each website. For each page $i$ of the $n$ pages explored at a website, the static analysis solution points-to pairs comprise set $S_i$ and the blended analysis solution points-to pairs, set $B_i$. Then the percentages shown in column 2 in Table III are:

$$\frac{\sum_1^n \frac{|S_i \cap B_i|}{|S_i|}}{n} \times 100\%$$

The average coverage of the static solution achieved by the blended analysis varies from 78.0% to 93.0%. The overall average (i.e., 86.6%) indicates that our blended analysis performed well on a large number of the programs. For 64 of the 709 webpages analyzed, blended analysis obtained 100% of

**TABLE II:** Benchmarks. Each benchmark is formed from an user interaction with a website that may explore one or more webpages. A profiled interaction consists of individual traces, each containing a sequence of JavaScript instructions from one webpage. The set of traces corresponding to the same webpage comprises a JavaScript program analyzed by our framework.

| Website | Page count | Trace count | *eval* page | Variadic func. | Website | Page count | Website | Trace count |
|---|---|---|---|---|---|---|---|---|
| google.com | 203 | 2104 | 52 | 177 | bing.com | 19 | conduit.com | 16 |
| facebook.com | 138 | 1098 | 23 | 65 | twitter.com | 14 | imdb.com | 18 |
| youtube.com | 122 | 579 | 19 | 29 | linkedin.com | 12 | myspace.com | 24 |
| yahoo.com | 52 | 265 | 21 | 13 | qq.com | 18 | sohu.com | 19 |
| baidu.com | 49 | 147 | 6 | 16 | wordpress.com | 23 | xing.com | 18 |
| wikipedia.org | 67 | 130 | 0 | 3 | sina.com.cn | 16 | xunlei.com | 22 |
| live.com | 54 | 226 | 10 | 44 | 163.com | 22 | zedo.com | 16 |
| blogger.com | 24 | 146 | 6 | 7 | cnn.com | 18 | washingtonpost.com | 27 |
| **totals** | **709** | **4695** | **137** | **354** | msn.com | 18 | pconline.com.cn | 21 |

the static points-to solution. For 36.7% of the pages, blended analysis covered at least 95% of the pure static points-to solution. Nevertheless, for some individual webpages the coverage was low. For example, for in one page of *facebook.com*, blended analysis produced only 11% of the points-to pairs found by static analysis. This happened because we only used two traces of this webpage achieving method coverage of 21.2%; in contrast, the average method coverage per webpge over the 138 *facebook.com* pages was 91.2%.

To evaluate how well the *Trace Selector* performs, we ran the blended points-to analysis on all 4695 collected traces. The results show that, with the *Trace Selector* turned on, an overall average of 1.6% fewer points-to pairs were covered, while 21.7% fewer traces were analyzed.

Table III also illustrates the effect of dynamically generated/loaded code on the points-to solution. In column 3 we show the average per webpage of the number of additional points-to pairs reported by blended analysis for each website, as a percentage of the entire program solution (i.e., $S_i \cup B_i$ for page $i$). Assume that $k$ pages in a website contain dynamic code. Then the results shown correspond to:

$$\frac{\sum_1^k \frac{|B_i - S_i|}{|S_i \cup B_i|}}{k} \times 100\%$$

There were 137 pages (19% of total pages observed) containing dynamically generated code by $eval$ and 383 pages (more than 50%) containing new JavaScript code dynamically loaded at runtime.[6]

The distribution of dynamic pages differed across websites. For *wikipedia.org*, we were not able to observe any calls to $eval$ or run-time loaded code; therefore, no additional points-to pairs were found by blended analysis. For other sites, blended analysis added between 1.4% (*blogger.com*) and 9.9% (*youtube.com*) points-to pairs not found by pure static analysis. The results suggest that dynamically generated/loaded code can not be ignored when analyzing the behavior of JavaScript programs, a result which concurs with the observation of dynamic code loading in [20].

[6]We observed that often for initialization the same piece of library code may be loaded across all the webpages in a website. In order to avoid the parts of the points-to solution due to this shared code dominating our results or when the corresponding code is lengthy, dominating the algorithm's performance, we analyzed such code for only one webpage in the website.

**Pruning & Variadicity.** Pruning was applied to the 3677 traces analyzed in the static phase. On average over the traces, 25.1% of the basic blocks (i.e., statements) were pruned. Removing these statements means the *Static Analyzer* had less code to analyze for each trace so that it ran faster. We found 354 variadic functions in total (Table II). We observed that in these functions $arguments.length$ is often used in a branch or loop condition. Among all the variadic functions we observed, we were able to prune 34 of them because of branch conditions containing $arguments.length$.

**TABLE IV:** Analysis time comparison (in minutes)

| Website | Static | Blended | | | |
|---|---|---|---|---|---|
| | | dynamic phase | static phase | total | slowdown |
| google.com | 78.4 | 13.1 | 92.0 | 105.1 | 34.1% |
| facebook.com | 116.8 | 11.7 | 120.6 | 132.3 | 13.3% |
| youtube.com | 49.3 | 8.8 | 67.2 | 76.0 | 54.2% |
| yahoo.com | 41.0 | 6.6 | 50.1 | 56.7 | 38.3% |
| baidu.com | 28.4 | 3.6 | 32.2 | 35.8 | 26.0% |
| wikipedia.org | 20.3 | 3.7 | 22.2 | 25.9 | 27.6% |
| live.com | 32.8 | 4.8 | 31.3 | 36.1 | 10.1% |
| blogger.com | 10.2 | 3.6 | 11.6 | 15.2 | 49.0% |

**Analysis Time.** Table IV presents the time performance of blended points-to analysis on average over the executed pages on each website. We compare the analysis time of the pure static and blended analyses. The analysis time of blended includes the time for the Dynamic Phase and Static Phase on each trace. Blended analysis imposes a slowdown of 31.6% on average compared to pure static analysis across all websites. (Recall that pure static analysis only analyzes statically accessible JavaScript code.) In a blended analysis, the static analysis phase, performed off-line, is more costly than execution of the lightweight instrumentation online. The main reason for the comparative slowness of blended analysis versus pure static is that some page traces may overlap in method coverage, causing those methods to be statically analyzed multiple times in our current implementation. The heuristic criterion in the *Trace Selector* is designed, in part, to prevent this situation. Blended points-to analysis analyzing all the traces executed 43.5% slower than with the *Trace Selector* turned on.

**stmt-level MOD Analysis for Debugging** We implemented a stmt-level MOD client of blended points-to analysis for

**TABLE V:** Debugging client using points-to information

| Website | Static | Blended | |
|---|---|---|---|
| | | objects in static code | objects in dynamic code |
| google.com | 5.8 | 2.4 | 2.1 |
| facebook.com | 7.7 | 4.1 | 3.6 |
| youtube.com | 5.9 | 3.5 | 2.4 |
| yahoo.com | 5.2 | 2.5 | 2.8 |
| baidu.com | 2.6 | 1.4 | 1.8 |
| live.com | 2.9 | 1.6 | 2.2 |
| blogger.com | 4.5 | 2.8 | 2.3 |

**TABLE VI:** Tainted input analysis results

| Website | Static | | Blended | |
|---|---|---|---|---|
| | true alarm | false alarm | true alarm | false alarm |
| live.com | - | - | 1 | - |
| youtube.com | 1 | - | 1 | - |
| myspace.com | - | - | 1 | - |
| sohu.com | 2 | 1 | 2 | - |
| xunlei.com | 3 | - | 3 | - |
| msn.com | - | - | 1 | - |
| bing.com | - | 1 | - | - |
| **totals** | **6** | **2** | **9** | **-** |

JavaScript (Section III-C); the analysis solution for this client is directly applicable to debugging. Blended analysis of a set of buggy executions can provide those JavaScript properties which experienced side effects on those executions. This information can be explored by a programmer to find unexpected side effects which may have contributed to the bug.

In this experiment, we used the blended points-to solution to solve stmt-level MOD on one page trace from each website, omitting *wikipedia.org* because its executions did not contain dynamic code, (i.e., seven traces in total). Each page trace was selected to contain the maximal number of methods observed for that website. The data in columns 2 and 3 in Table V show the results for the shared static code (i.e., statically visible code executed by the trace, and thus analyzed by both blended and pure static analysis). There were on average 46.7% fewer objects referenced in the blended analysis results, which means a programmer had to examine only about half the objects, (i.e., those which may have experienced side effects on the buggy executions). This information focuses programmer attention on possible causes of the bug. Column 4 shows the results in the dynamically generated/loaded code, which may contain the root cause of bugs.

**Summary.** The empirical results on client I show that blended points-to analysis is a promising technique for generating points-to information for JavaScript programs. It achieved good solution coverage and produced additional results from dynamic code at reasonable cost, supporting *Hypothesis 1*. Given that the current implementation is a research prototype, we believe that blended analysis performance still can be improved greatly. In our study of stmt-level MOD analysis, we showed that blended analysis can provide more accurate, focused information for debugging, thus supporting *Hypothesis 2*.

### C. Client II: Tainted Input Analysis

We compared our blended tainted input analysis with a pure static tainted input analysis. The pure static analysis uses the *WALA* provided points-to analysis to generate the call graph as an input to tainted input analysis. Because the static analysis in *WALA* does not model the $eval$ construct, the pure static tainted input analysis has to consider $eval$ statements to be sinks.

Table VI shows the results of static and blended tainted input analyses for JavaScript on all 26 websites in Table II. The total number of actual reports was 30 (11 were false alarms) for static analysis and 31 for blended analysis. We report the number of unique alarms in Table VI by only counting once those alarms from the same website; these alarms originated from the same cloned JavaScript code on different webpages. Pure static analysis reported 8 *unique* source-sink pairs from 4 of the 26 websites; 2 of them were false alarms. Blended tainted input analysis reported 9 unique pairs, all of them true positives, from 6 websites. We have manually examined this data and concluded the following observations.

Pure static analysis reported one source-sink pair from *bing.com* and three pairs from *sohu.com*; however, blended analysis reported 0 and 2 pairs, respectively. We examined these cases and found that two static analysis reports were false alarms because static analysis did not have access to the code inside the $eval$ construct. For instance, the false alarm of *sohu.com* originated from the function *changeid(id)* where $id$ is tainted reaching the sink *eval("obj" + classrand)* where $classrand$ is tainted. The code executed did not flow from a tainted input into an actual sensitive sink as defined in section III-C so blended analysis did not report this flow.

Our blended tainted input analysis was able to find three additional source-sink pairs in three websites (*live.com*, *myspace.com* and *msn.com*). This was because the sinks were located in the dynamically generated/loaded code which static analysis did not analyze.

For the remaining two websites (*youtube.com* and *xunlei.com*), blended and static tainted input analyses reported the same results which all were verified to be true positives.

**Summary.** Blended analysis reported all 6 true positives found by static analysis, reported no false alarms, and also found three additional true positives. Over all the webpages that contained tainted source-sink pairs reported by either analysis, blended analysis analyzed only on average 62.5% of the methods in the static code. Thus, even with fewer methods analyzed, our blended analysis achieved the same results as static analysis, and actually found additional true positives, supporting *Hypothesis 3*.

### D. Threats to validity

There are several aspects of our experiments which might threaten the validity of our conclusions: (i) The accuracy of our implemented framework is determined by limitations of the *WALA* interpretation of JavaScript. We found that there were some parsing problems with some JavaScript code in the websites, and some structures of JavaScript were ignored.

(ii) Our experiments made comparisons between static points-to analysis in *WALA* and static tainted input analysis we implemented ourselves. These static analyses do not handle dynamically generated/loaded codes. Other existing static analyses of JavaScript may provide better solutions for some of its dynamic features (see section V); however, there still are some uses of reflective constructs which cannot be handled by them. We hope to compare to some of these newer approaches in future work. (iii) Because we collected the executions of websites ourselves, we may have introduced a bias in terms of the pages explored. To avoid this as much as possible, one author collected executions without knowledge of the JavaScript website code. (iv) Although we used websites listed at *Alexia* as most popular, we cannot know how representative our input is of normal website usage. (v) Although these initial experiment results are promising, more empirical investigation will be necessary for a stronger validation of our hypotheses.

## V. Related Work

In this section, we present work related to our JavaScript blended analysis. Due to space limitations, we focus only on the most relevant research: (i) blended analysis of Java; (ii) studies of JavaScript dynamic behavior; (iii) static analyses that try to facilitate the handling of some dynamic language features of JavaScript; (iv) hybrid analyses of JavaScript.

**Blended analysis of Java.** Blended analysis of Java [4], [5] performed an interprocedural static analysis on an annotated program calling structure obtained by lightweight profiling of a Java application, focusing on *one* execution that exhibited poor performance. The annotations recorded the observed call targets and allocated types for each executed method, information that enabled pruning of a significant number of unexecuted instructions.

Blended analyses for Java and for JavaScript both apply a dynamic analysis followed by a static analysis on the collected calling structure and both use pruning based on dynamic information. However, Java blended analysis focuses on one problematic execution, while JavaScript blended analysis analyzes a set of executions appropriate to the problems. The complexity of dynamic analysis for JavaScript far exceeds that in the Java analysis. The latter merely records all calls, including reflective ones. The former captures dynamically generated/loaded code and records all calls therein, a more difficult task especially with nested reflective constructs (e.g., *evals*). The JavaScript pruning uses a richer set of dynamic information than is used in Java pruning. Thus, while the blended algorithms are related as to overall high-level structure, there are many differences between them and the dynamic language constructs analyzed are very different.

**Dynamic behavior.** The dynamic behavior of JavaScript applications reflects the actual uses of dynamic features. Richards, *et al.* conducted an empirical experiment on real-world JavaScript applications, (i.e., websites), to study their dynamic behavior [21]. The behaviors studied include call site dynamism, function variadicity, object protocol dynamism, etc. The authors concluded that common static analysis assumptions about the dynamic behavior of JavaScript are not valid. Our work is motivated by their study. Studies of the uses of $eval$ in JavaScript applications [20] show that the $eval$ construct, which can generate code at runtime, is widely used. This result justifies our emphasis on $eval$ in blended analysis.

Ratanaworabhan, *et al.* also presented a related study on comparing the behavior of JavaScript benchmarks, (e.g., *SunSpider* and *V8*), with real Web applications [18]. Their results showed numerous differences in program size, complexity and behavior which suggest that the benchmarks are not representative of JavaScript usage. This study motivated us to evaluate our blended points-to analysis on website codes.

**Static analysis.** Various static analyses have been applied to JavaScript. Guarnieri, *et al.* [9] presented ACTARUS, a pure static taint analysis for JavaScript. Language constructs, including object creations, reflective property accesses, and prototype-chain property lookups were modeled, but reflective calls like $eval$ were not modeled. Our *Static Analyzer* uses the same *WALA* infrastructure so that we share some models for JavaScript constructs in common with this work. Jang and Choe [12] presented a static points-to analysis for JavaScript. This context- and flow-insensitive analysis works on SimpleScript, a restricted subset of JavaScript, (e.g., prototyping not allowed). Guarnieri and Livshits presented another static points-to analysis to detect security and reliability issues and experiment with JavaScript widgets [7]. JavaScript$_{SAFE}$ is a subset of JavaScript that static analysis can safely approximate, even with reflective calls such as $Function.call$ and $Function.apply$. Other forms such as $eval$ are not handled. None of the above JavaScript static analyses can model all of the language's dynamic features, (e.g., *eval*), whereas our analysis framework can handle most of the more common dynamic features used by real websites.

Jensen, *et al.* [13] applied static analysis to automatically transform some $eval$ calls into other language constructs. The results show that their approach is able to eliminate typical uses of nontrivial $eval$ although there are cases where the technique has to give up. Our blended analysis for JavaScript shares the same goal as this work. We expand the applicability of static analysis more generally by recording the actual code generated in the *Dynamic Phase*.

Several type-based analyses have been proposed [14], [11], [1], [15]; however, JavaScript's dynamism makes it hard to achieve good precision. These approaches work on subsets of JavaScript (e.g., $JavaScript^{=}$ [15] and $JS_0$ [1]) and were evaluated on JavaScript benchmarks, (e.g., Google *V8* and *SunSpider*), rather than website code. Blended analysis, on the other hand, analyzes real website codes.

**Hybrid analysis of JavaScript.** Several staged analyses of JavaScript, analyze the statically visible code first and then incrementally analyze the dynamically generated code. Chugh, *et al.* [3] presented an information flow analysis for JavaScript. Guarnieri and Livshits [8] provided GULFSTREAM as a staged points-to analysis for streaming JavaScript applications. JavaScript blended analysis differs from their approaches in

two ways. Blended analysis collects dynamically generated/loaded code during profiling rather than doing this incrementally. Blended analysis also facilitates potentially more precise modeling of other dynamic features whose semantics depend on run-time information.

Vogt, *et al.* presented a hybrid approach to prevent cross-site scripting [17]. In this work, dynamic taint analysis tracks data dependencies precisely and static analysis is triggered to track control dependencies if necessary. Trace-based compilation for JavaScript [6], [10] focused on performance issues. It is a hybrid approach in terms of dynamic trace recording and applying simple static analyses on specialized traces. Blended analysis is different from these approaches as a flexible, general-purpose analysis framework on which we can build all kinds of client applications (e.g., for security and optimization).

## VI. Conclusion

JavaScript is widely used as a programming language for client-side Web applications. Analyzing JavaScript programs statically is difficult because its dynamic features cannot be precisely modeled. The JavaScript Blended Analysis Framework is designed to address these challenges. We implemented two clients for blended analysis, points-to analysis with stmt-level MOD and tainted input analysis. Blended points-to analysis covered 86.6% of the pure static points-to solution on average and added an average of 7.0% of the points-to pairs to the actual points-to solution, at a cost of 31.6% slowdown over pure static analysis. Blended stmt-level MOD resulted in 46.7% fewer objects referenced to reduce the programmer's effort needed for debugging. Blended tainted input analysis reported all 6 true positives found by pure static analysis, but without any false alarms; blended analysis also managed to find three additional true positives on dynamically generated/loaded code. Thus, the experimental results show that blended analysis is a practical and flexible approach for analyzing JavaScript programs, even those which use dynamic constructs such as $eval$.

In future work, we plan to improve our blended analysis by providing more precise models for other JavaScript features, such as prototyping. We also would like to explore alternatives in our blended analysis framework (e.g., using different calling structure representations). We also are looking for other clients that naturally fit with our blended analysis.

## References


[1] C. Anderson, S. Drossopoulou, and P. Giannini. Towards type inference for JavaScript. In *19th European Conference on Object-Oriented Programming (ECOOP 2005)*, pages 428–452, June 2005.

[2] A. Barth, J. Weinberger, and D. Song. Cross-origin javascript capability leaks: detection, exploitation, and defense. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 187–198. USENIX Association, 2009.

[3] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 50–62. ACM, 2009.

[4] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 118–128. ACM, 2007.

[5] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 59–70. ACM, 2008.

[6] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478. ACM, 2009.

[7] S. Guarnieri and B. Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 151–168. USENIX Association, 2009.

[8] S. Guarnieri and B. Livshits. Gulfstream: staged static analysis for streaming javascript applications. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, pages 6–6. USENIX Association, 2010.

[9] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 177–187. ACM, 2011.

[10] S.-y. Guo and J. Palsberg. The essence of compiling with traces. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 563–574. ACM, 2011.

[11] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 200–224. Springer-Verlag, 2010.

[12] D. Jang and K.-M. Choe. Points-to analysis for javascript. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1930–1937. ACM, 2009.

[13] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 34–44. ACM, 2012.

[14] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255. Springer-Verlag, 2009.

[15] F. Logozzo and H. Venter. Rata: Rapid atomic type analysis by abstract interpretation - application to javascript optimization. In *CC*, volume 6011 of *Lecture Notes in Computer Science*, pages 66–83. Springer, 2010.

[16] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: a unified model. *Acta Inf.*, 28:121–163, December 1990.

[17] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2007.

[18] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. Jsmeter: comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, pages 3–3. USENIX Association, 2010.

[19] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of javascript benchmarks. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 677–694. ACM, 2011.

[20] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 52–78. Springer-Verlag, 2011.

[21] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12. ACM, 2010.

[22] S. Wei and B. G. Ryder. A practical blended analysis for dynamic features in javascript. Technical Report TR-12-11, Virginia Tech, 2012.