Practical Blended Taint Analysis for JavaScript

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

ABSTRACT

JavaScript is widely used in Web applications because of its flexibility and dynamic features. However, the latter pose challenges to static analyses aimed at finding security vulnerabilities, (e.g., taint analysis).

We present *blended taint analysis*, an instantiation of our general-purpose analysis framework for JavaScript, to illustrate how a combined dynamic/static analysis approach can deal with dynamic features by collecting generated code and other information at runtime. In empirical comparisons with two pure static taint analyses, we show blended taint analysis to be both more scalable and precise on JavaScript benchmark codes extracted from 12 popular websites at *alexa*. Our results show that blended taint analysis discovered 13 unique violations in 6 of the websites. In contrast, each of the static analyses identified less than half of these violations. Moreover, given a reasonable time budget of 10 minutes, both static analyses encountered webpages they could not analyze, sometimes significantly many such pages. Case studies demonstrate the quality of the blended taint analysis solution in comparison to that of pure static analysis.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.3.4 [**Programming languages**]: Processors

General Terms

Languages, Security, Design

Keywords

JavaScript, program analysis, taint analysis

1. 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 (*www.alexa.com*) use JavaScript [8]. 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. Often dynamic constructs provide opportunities for security exploits.

Recent studies [16, 19, 18] reveal that JavaScript programs are full of dynamic features, and that the dynamic behavior of actual websites confirms this fact. There are several mechanisms in JavaScript whereby executable code can be generated at runtime (e.g., *eval*). Richards *et al.* [18] 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 well statically. In Richards *et al.* [19], 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 statically about JavaScript applications. Existing static analysis approaches cannot handle the most general uses of these constructs (see Section 6).

Given the ubiquity of JavaScript, it is crucial to discover security vulnerabilities possibly introduced by use of its dynamic features. Websites usually contain both user interaction and third party components. If these components are untrusted, then they may be used to exploit vulnerabilities. For example, a malicious user (or package) might inject a input string value that can be parsed into executable JavaScript code which, if not properly sanitized, can cause a cross-site scripting attack.

Many security problems can be formalized as information flow problems [3] which seek to preserve the integrity of data (i.e., not allow untrusted values to affect a sensitive value or operation) and confidentiality of data (i.e., keep sensitive values from being observed from outside the computation). Taint analysis detects flows of data that violate program integrity. Several important security vulnerabilities can be identified using taint analysis (e.g., cross-site scripting, SQL injection) [8]. Some analysis approaches have been proposed to detect and/or prevent such security vulnerabilities in JavaScript [1, 2, 9, 15]. Nevertheless, there is room for improvement in JavaScript taint analysis for real-world applications.

We have built a new JavaScript Blended Analysis Framework (*JSBAF*) to investigate how to design a practical analysis for a general-purpose scripting language while accommodating its dynamic features. The framework is flexible in that individual components are independent and substitutable. Our JavaScript blended analysis captures rich information about dynamic language features. These include dynamically generated (or loaded) JavaScript code (e.g., through *eval* functions or interpreted *urls*) and variadic function usage. Because of the wide-spread usage of these dynamic features, their capture is important. A pure static analysis [12] may miss them (e.g., when an *eval* contains a JavaScript code string which contains user input) 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).

Other dynamic languages like PHP, Ruby, and Perl share several dynamic features with JavaScript, (e.g., dynamic types, run-time code generation). We believe our approach can be applied to the analysis of these languages as well; we will investigate that in future work.

The focus of this paper is our instantiation of *JSBAF* to perform blended taint analysis for JavaScript. We present an empirical comparison of our blended results with two different pure static analysis approaches. The experimental results demonstrate the practicality of our approach, and its scalability and precision with respect to static analysis on benchmark codes from 12 of the top 25 websites on *www.alexa.com*. Less than half of the 13 true exploits found by our blended analysis were identified by either of the static analyses. Moreover, blended analysis reported only one false positive vulnerability (i.e., false alarm). Additionally, the static phase of our blended analysis ran to completion under a limited time budget of 10 minutes whereas one pure static analysis failed to complete on 41% of the webpages analyzed.

The major contributions of this paper are:

Blended Taint Analysis for JavaScript. A new practical and accurate blended taint analysis for JavaScript that is an instance of *JSBAF*.

Empirical results. Our empirical results show the relative strength in scalability and precision of blended taint analysis for JavaScript in comparison to a pure static taint analysis built with a state-of-the-art points-to analysis [22] and an alternative library-free pure static analysis. Case studies are presented from actual website codes to illustrate specific cases where the quality of blended analysis results over pure static analysis results is demonstrated.

Overview. The rest of this paper is organized as follows: Section 2 uses an example to illustrate the dynamism of JavaScript. Section 3 introduces blended analysis of JavaScript and our framework, *JSBAF*. Section 4 describes the instantiation of *JSBAF* for taint analysis and gives details of the analysis components used in our implementation. Section 5 presents our experimental results, Section 6 discusses related work, and Section 7 offers conclusions and future work.

2. MOTIVATING EXAMPLE

In this section, we present a sample HTML webpage containing a JavaScript program to illustrate the challenges of analyzing dynamic languages. The program in Figure 1 incorporates Web/JavaScript features such as function variadicity, an asynchronous HTTP request, *eval* and path-dependent dynamic dispatch. The purpose of this example is to give intuition as to how blended analysis works.

In Figure 1, the JavaScript code within the `<script>` tags is loaded when the HTML page opens. A form named *frm* (lines 24–27) contains two text `<input>` elements designed to receive the values username (*usr*) and password (*pwd*) for the login/signup procedure.

```
1  <html><body>
2  <script>
3  function foo(){
4    var pwd = document.forms["frm"]["pwd"];
5    if (pwd == "")
6      goo(document.forms["frm"]["usr"]);
7    else goo(document.forms["frm"]["usr"], pwd);}
8  function goo(){
9    if(arguments.length == 1)
10     var data = "signin:" + arguments[0];
11   else
12     data ="singup: "+arguments[0]+" "+arguments[1];
13   var xmlhttp = new XMLHttpRequest();
14   xmlhttp.onreadystatechange=function(){
15     if (xmlhttp.readyState==4){
16       if(eval(xmlhttp.responseText) == "sign-in")
17         document.getElementById("div1").innerHTML
18           =xmlhttp.responseText;
19       else document.write(xmlhttp.responseText);
20     }}
21   xmlhttp.open("POST", url ,true);
22   xmlhttp.send(data);}
23 </script>
24 <form name="frm" onsubmit="foo">
25 <input type="text" name="usr">
26 <input type="text" name="pwd">
27 <input type="submit" value="Submit"></form>
28 <div id="div1"></div>
29 </body><html>
```

Figure 1: A login/signup procedure webpage

The function foo (lines 3–7) will be called when the *submit* button in the HTML form (line 27) is clicked (i.e., the *onsubmit* event is triggered (line 24)). foo performs a check (line 5) to call function *goo* with either one (when *pwd* is empty) or two arguments.

The function signature of *goo* (line 8) is written with no argument; however, *goo* is designed to be called with different numbers of arguments. In lines 9–12, the two branches of the if statement assign different values to the variable *data* depending on the length of argument list (i.e., *arguments.length*). An asynchronous HTTP request to the server is generated (lines 13–22). The client data is sent to the server (line 22) *via*. the POST method. In lines 14–20, the $responseText$ of the request changes the HTML Document Object Model (DOM) through either $innerHTML$ of the $div1$ element (lines 17–18) or the *document.write* function (line 19). The predicate in line 16 involves an invocation to *eval* which generates and evaluates the code in *xmlhttp.responseText* at runtime. The *eval* is needed here because the $responseText$ is a string of code received from the server. We observed cases using *eval* in the same pattern (e.g., in *www.bing.com*).

Consider the example in Figure 1 in the context of finding security violations. We focus our paper on security vulnerabilities of JavaScript programs in client-side Web applications. User inputs are not trusted by the program because there could be attackers that act like legitimate users. Hence, user inputs should be not allowed to be stored on the server or to modify sensitive properties of the DOM, unless they are properly sanitized. The user inputs in the example are the values of elements *usr* and *pwd*. To illustrate the complexity of an analysis to detect security violations, assume that the value of *usr* is sanitized, which leaves the value of *pwd* as the only tainted source. *foo* can invoke *goo* at two call sites (lines 6-7), one of which uses the tainted source as a parameter (line 7). It is hard for a static analysis, (e.g., object-sensitive analysis [14]), to distinguish these variadic function calls. In *goo*, only one branch generates the *data* variable that is tainted (line 12) and sends it to the server (line 22).

It is also difficult for a static analysis to analyze the statement in line 16 because the dynamic code generated by *xmlhttp.responseText* cannot be seen by a pure static analysis. This generated code may propagate tainted sources to affect sensitive sinks in the program; thus, it must be analyzed. Also, this program contains implicit semantic relations between the branches of the if statements in *goo* at lines 9 and 16 (e.g., if *pwd* is empty, only *div*1 will be modified; otherwise, the *document* will be written by the response text). Discovering these relations requires a path sensitive analysis, which may be too costly. A static analysis will merge the solutions on those branches; thus imprecisely will report security violations (e.g., *div*1 will be reported as containing sensitive data).

Blended analysis offers an approach to handle these issues. Dynamically generated code is observed by blended analysis so that the *eval*(*xmlhttp.responseText*) is captured as the actual code executed. A dynamic call tree also is captured providing the number of arguments at each call site executed. Blended analysis separately analyzes function instances called with different numbers of arguments. For example, the call sites in lines 6 and 7 are distinguished in blended analysis because the same function *goo* is called with 1 and 2 arguments, respectively. By retaining some execution path information, blended analysis prunes away some of the unexecuted program (see Section 3). For example, if *pwd* is empty then the else branches of all the three conditionals will not be executed. Blended analysis prunes these branches and thereby eliminates the imprecise approximation that the tainted source *pwd* can flow into *div*1.

3. BLENDED ANALYSIS OF JAVASCRIPT

In previous work, we designed a blended analysis for performance diagnosis of framework-intensive Java programs. This analysis dynamically collected a problematic Java execution and performed a static escape analysis on its calling structure [4, 5]. 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 [12]). Intuitively, Java blended analysis focused a static analysis on a dynamic calling structure collected at runtime, and further refined the static analysis using additional information collected by a lightweight dynamic analysis.

Pruning was an optimization technique applied in Java blended analysis to each executed method's control flow graph in order to approximate a specialized version of the code executed during a particular call. Pruning was very effective in removing approximately 30% of the statements from Java functions [5]. Essentially, using run-time information we removed unexecuted statements in functions by noticing which function calls and object creation sites were not recorded and by using control dependence information.

The blended algorithm paradigm of tightly coupled dynamic and static analyses is the basis of our analysis for JavaScript. We analyze multiple executions rather than a single one, but the overall algorithm workflow and pruning are both utilized, albeit to handle a more general set of dynamic language features in JavaScript. We have designed a general-purpose blended analysis framework for JavaScript, *JSBAF*, shown in Figure 2. In this section we present an overview of this flexible framework and explain how it addresses several challenges of analyzing JavaScript programs. We also discuss the dynamic features of JavaScript which can be accommodated by *JSBAF*.

3.1 JSBAF

JSBAF was designed to judiciously combine dynamic and static analyses in a practical but unsafe [12] analysis of JavaScript, to account for the effects of dynamic features not seen by pure static analysis, while retaining sufficient accuracy to be useful.

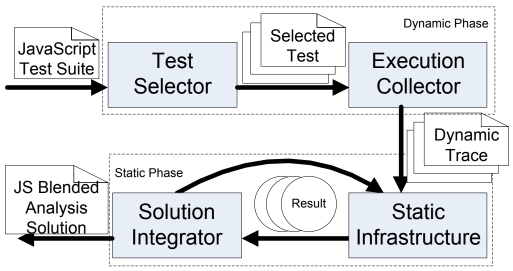

Figure 2: JavaScript Blended Analysis Framework

As shown in Figure 2, *JSBAF* can be applied in the following software testing scenario for a JavaScript program. We assume the presence of a good test suite providing program coverage information along with each test, which is usually available in enterprise software. The *Test Selector* chooses a subset of the tests that offer good method coverage of the program to obtain a good analysis solution at lower cost than using all the tests.[1] The *Execution Collector* gathers run-time information by executing each selected test, (e.g., method calls, dynamically generated code). The *Static Infrastructure* analyzes the program represented by the observed calling structure. The *Solution Integrator* combines dataflow solutions from different test traces into a program solution, and decides if there are more traces to analyze.

By design, *JSBAF* performs static analysis on each dynamic trace and combines the results. An alternative design would be to combine the dynamic information from all the executions and then apply static analysis once to the combined executions. While this alternative approach may save the cost of multiple static analyses, we believe it also may lose precision by introducing possibly infeasible interprocedural paths. Our experimental results (see Section 5.2.2) demonstrate the high accuracy of our blended analysis which analyzes traces individually.

JSBAF has a flexible workflow in which individual components can be substituted. For example, by changing the

[1]Note: in Section 4 we present a blended taint analysis that does not require an existing test suite with coverage information. In this context, we are still able to provide useful analysis results at practical cost *via.* a slight variation of the framework.

instrumentation heuristics of the *Execution Collector*, we can collect different dynamic information. By replacing the *Static Infrastructure*, we can change the specific analysis applied to the JavaScript program.

3.2 Dynamic Features in JavaScript

3.2.1 *eval*

Pure static analysis can analyze JavaScript source code that is statically visible; 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 *eval* parameters may contain values set at runtime. Dynamic code generation mechanisms make static analysis unsafe when analyzing JavaScript programs. Recently, Jensen *et al.* presented a static analysis technique to extract and transform JavaScript code from *eval* at compile time, thus enabling its static analysis [10]. However, as we show in one of our case studies, there are inevitably cases where this approach cannot transform some calls to *eval*.

```
for (n = 1; n < 20; n++) {
    xe = "s.prop" + n + "=myUe(s.prop" + n + ")";
    ex = "s.eVar" + n
         + "=myCp(s.prop" + n + ",'D=c" + n + "')";
    to = "typeof(s.prop" + n + ")";
    if (eval(to) != "undefined") {
        eval(xe);
        eval(ex)
    }
}
```

Figure 3: JavaScript *eval* example

Figure 3 shows an example of *eval* functions from *xing.com*, whose complex code contains *eval* calls that cannot be analyzed by the approach in [10]. In *JSBAF*, *eval* calls are monitored by the *Execution Collector* which gathers any code generated thusly, making it available during analysis of the dynamic calling structure. The *Static Infrastructure* analyzes the JavaScript program including the effects of *evals*. In addition to *eval*, there are other mechanisms in JavaScript such as *Function* object and *SetInterval* function that also can generate code at runtime [18]; these also are collected and analyzed by blended analysis. Blended analysis for JavaScript includes the effects of dynamically generated code whose run-time behavior is captured by the *Execution Collector*.

3.2.2 Function Variadicity

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* array. Sometimes, branch conditions within a function can be differentiated by its number of arguments as in the *goo* function of Figure 1.

Figure 4 shows a real use of variadic function. The code is extracted from *www.linkedin.com*. The arguments calling J vary depending on the number of arguments provided when this function is called, (i.e., *evt*:*null* & L:*arguments*[0] if one argument is provided; *evt*:*arguments*[0] & L:*arguments*[1] if two arguments are provided). This suggests the function

```
function(){
    if(arguments.length===1){
        evt=null;
        L=arguments[0]
    }else{
        evt=arguments[0];
        L=arguments[1]
    }
    J(evt,L)
}
```

Figure 4: JavaScript variadic function example

behavior of J varies based on the *arguments.length*. Existing pure static analyses for JavaScript normally ignore this feature because sometimes the actual number of arguments provided during the call can only be known at runtime (e.g., the calls to a variadic function f may be written as *f.apply(thisArg, argsArrary[])* using an argument array *argsArray* whose length is not knowable statically). In contrast, the *Execution Collector* can capture the actual number of arguments for each call so that the calling structure can contain separate nodes for instances of the same signature function called with different numbers of arguments, introducing some context sensitivity [21]. In addition, pruning provides a more precise model of the code in variadic functions, differentiating between invocations with different numbers of arguments (see Section 4.2).

3.2.3 Other Features

In addition to the two important dynamic features we handled in *JSBAF*, there are other features in JavaScript that require special treatment. Some features, (e.g., prototypes, object creations, reflective property access and lexical scoping), were modeled by the static analysis in [8], but the precision of these static models can be improved.

```
1  if (b) {
2    x = new A();
3  else
4    x = new B();
5  }
6  x.bar();
```

Figure 5: JavaScript type-based dynamic dispatch example

JavaScript is a dynamically typed programming language; hence, statically reasoning about object types is a big challenge. Figure 5 illustrates an example of dynamic dispatch based on the object type. Because of the nature of dynamic typing, the variable x can point to objects whose types are unrelated by inheritance (e.g., the objects created by constructors A and B). The actual function being called in line 6 depends on the type of x, which is determined by the value of b in line 1. In *JSBAF*, the *Execution Collector* collects the functions that are called and constructors that are used to create JavaScript objects. We use pruning to eliminate the code that was not executed to preserve the dynamic information. In case of Figure 5, one of the branch (e.g., line 4) will be pruned if x is created only by A so that the blended analysis knows the actual type of x in line 6. Thus blended analysis achieves some context sensitivity in analyzing a dynamic trace. In contrast, static analysis must assume that either branch can be taken, and thus make a conservative approximation that x can be either an A or B type object, making the target in line 6 ambiguous. Static analysis effectively merges the two paths to obtain a less precise solution.

4. BLENDED TAINT ANALYSIS

We have defined blended taint analysis to identify security vulnerabilities due to data integrity violations in JavaScript codes in websites. Figure 6 presents the workflow of our blended taint analysis, an instantiation of *JSBAF* (Figure 2). The work of the dynamic and static phases of blended taint analysis is patterned directly after the work of these phases in *JSBAF*. In the *Dynamic Phase*, a web tester interacts with a website using a browser that instruments JavaScript operations. Traces of each webpage consisting of recorded method calls, types (represented by the constructors) of created objects, and dynamically generated/loaded code that is not statically visible, are gathered by the *Execution Collector*. The *Trace Selector* selects a subset of the page traces that cover the behavior of the program well (see Section 4.1). In the *Static Phase*, the *Code Collector* identifies the JavaScript code that was executed, including both statically visible and invisible code. The *Call Graph Builder* creates a call graph from the recorded calls and stores other collected method-specific information as node annotations. *Static Taint Analysis* is applied to the program represented by the call graph. The *Solution Integrator* combines solutions from different page traces into a single solution for that webpage. The final solution of a blended taint analysis is a set of source-sink pairs reported on the webpages of the website being analyzed; these represent untrusted data (i.e., *sources*) which can reach sensitive object properties (i.e., *sinks*) (see Section 4.2).

4.1 Dynamic Phase

Our *Execution Collector* relies on a specialized version of *TracingSafari*, an instrumented version of WebKit[2] JavaScript engine developed for characterizing the dynamic behavior of JavaScript programs [19]. 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 source file loads. Since the dynamic phase of blended taint analysis involves programmer interaction, the instrumentation should degrade browser performance as little as possible. We modified *TracingSafari* to collect only the information required by our taint analysis.

To assure the security of a website, the web tester explores webpages from the same domain. 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 JavaScript instructions from the same webpage (i.e., *url*). There is at least one trace generated for each page executed containing JavaScript code. An interactive webpage with complicated functionality is usually executed more than once in a test session, so that it 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 only instructions recorded are calls and object allocations. The *Execution Collector* also captures the actual number of arguments for each call to precisely model variadic functions (Section 3.2.2). The *Trace Extractor* builds the set of page traces corresponding to each webpage collected from a set of recorded website executions.

There may be traces of the same page that are redundant on a large portion of their call trees; that is, even though the web tester tries to examine different components of a webpage, these components may vary little in terms of the JavaScript code executed. Blended taint analysis should avoid checking very similar traces since this will greatly increase the analysis cost for little benefit. To avoid this situation, we implemented the *Trace Selector* which tries to minimize the number of traces analyzed, while covering as much program behavior as possible. The *Trace Selector* calculates the information of each page trace and chooses traces that contribute the most to (1) dynamically loaded/generated code coverage, (2) method coverage and (3) created object type coverage. We developed metrics for each of these coverages and designed a heuristic combining them in a weighted average to select traces in order of greatest coverage with values above certain threshold. The straightforward selection process can be applied to other dynamic programming languages. (Please refer to [24] for details.) In future work, we want to implement an alternative *Trace Selector* that can generate traces on-the-fly and determine when enough traces have been explored.

4.2 Static Phase

The static infrastructure of our blended taint analysis for JavaScript was built on the *IBM T.J. Watson Libraries for Analysis (WALA)* open-source static analysis framework[3] 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* [8].

Our *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 have duplicate nodes in our *WALA* call graph, but each node context is different.

The *Call Graph Builder* applies pruning to the code of all functions. When branches of a variadic function are determined by the value of *arguments.length*, 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.

4.2.1 Static Taint Algorithm

The static taint algorithm we implemented to detect integrity violations consists of four steps.

[2]webkit.org

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

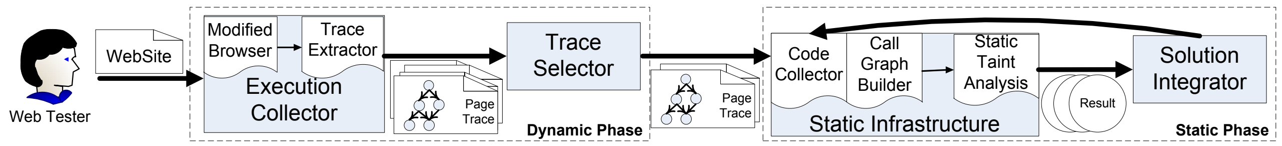

Figure 6: Blended Taint Analysis for JavaScript Web Applications

1. A state-of-the-art points-to analysis for JavaScript [22] is performed to obtain aliases of objects in the program. We modified the implementation by substituting use of our dynamic call graph for the on-the-fly construction.

2. Sources and sinks are predefined and automatically identified in the program:

(i) A data source is called *tainted* (or *untrusted*) when the user or an untrusted third party has control of its value. JavaScript event handlers that take user inputs as parameters or contain variables from user inputs are considered to be *sources*; their user input arguments and variables are marked as tainted. JavaScript functions from untrusted third party code are also considered to be *sources*; the variables created in these functions or whose values are returned by calling these functions are marked as tainted.

(ii) We consider two sets of objects to be *sensitive*. Fields of objects that hold important browser/user information are sensitive. In the implementation, we used the same set of data fields as in [15]. Every variable in a statement that writes those fields is marked as a sink. A persistent security vulnerability can happen if untrusted data is saved by the server. Therefore, parameters of functions which are sent to the server are sensitive (e.g, the parameter of *xmlhttprequest.send()*). These parameters are marked as sinks.

3. A call graph reachability analysis is executed to filter out any node that is not on a direct call path from a method containing tainted source(s) to a method containing sink(s); the remaining nodes are called *candidates*.

4. Taint propagation. An interprocedural traversal of the call graph is performed from each source through candidate nodes to any reachable sink. At each encountered candidate method, an intraprocedural data dependence analysis is applied to track the tainted variables into candidate calls. The possible effects of calls to non-candidate methods are approximated: if one argument of the call is tainted, we assume all the arguments are tainted as an optimization to avoid analysis of these methods. Call cycles are handled by fixed point iteration.

5. EVALUATION

We conducted experiments comparing the effectiveness of blended taint analysis with two pure static taint analyses. We present results from testing our analysis on JavaScript programs from popular websites and illustrate the advantages of blended taint analysis through case studies.

5.1 Experiment Design

5.1.1 Pure Static Taint Analyses

To compare with blended taint analysis, we implemented a pure static taint analysis in $WALA$ that performs the four-step algorithm presented in Section 4.2.1. Instead of using the dynamically collected call graph, the pure static taint analysis takes a statically visible JavaScript program as input and builds the call graph on-the-fly during points-to analysis [22].

The current static infrastructure in $WALA$ does not model the semantics of *eval*. To increase the capability of pure static taint analysis, we added a naive model of *eval*. JavaScript variables that are visible from *eval* parameters are collected and treated as accessible in the *eval* calls. In pure static taint analysis, the *eval* functions are conservatively marked as additional sinks because static analysis does not know their execution behavior.

In the experiment, we ran pure static taint analysis in two configurations. JavaScript libraries are usually loaded as provided by static *urls* in webpage source code. The first configuration, Static Taint$^+$, analyzes the code directly extracted from the webpages and any reachable library functions. The second configuration, Static Taint$^-$, analyzes only the JavaScript code extracted from the webpages. Because some JavaScript libraries are too large to analyze in limited time [22] and taint violations may originate solely from application code, we believed that Static Taint$^-$ would be capable of discovering some vulnerabilities without analyzing libraries.

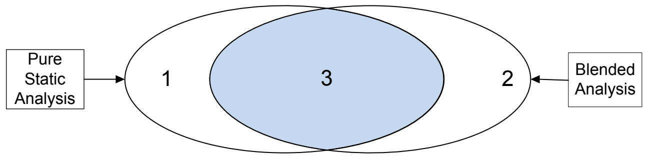

Figure 7: Relation between solutions obtained through pure static analysis vs. blended analysis

To understand our analysis comparison results, we describe the general relationship between a pure static analysis solution and a blended analysis solution for the same JavaScript program. For JavaScript, both static and blended analyses are unsafe. In Figure 7, 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 the difficulty of modeling dynamic constructs in JavaScript. It also may contain false positives because of the approximation in analysis of dynamic code. Part 3 of the diagram shows results reported by both analyses. The goal of blended analysis is to retain the true positives found by a pure static analysis while eliminating some false positives, and to find more true positives by analyzing the dynamic code.

Table 1: Benchmarks. Each benchmark is formed from a web tester's interaction with a website. A profiled interaction consists of individual traces, each containing a sequence of JavaScript instructions from a single webpage. The set of traces corresponding to the same webpage comprises a JavaScript program for our analysis.

Rank	Website	Page count	Trace count
1	facebook.com	27	62
2	google.com	22	55
3	youtube.com	15	30
4	yahoo.com	30	69
6	wikipedia.org	27	65
8	amazon.com	9	13
10	twitter.com	32	53
12	blogspot.com	9	17
14	linkedin.com	32	54
18	msn.com	13	21
19	ebay.com	40	72
21	bing.com	7	14
	totals	**263**	**525**

5.1.2 Hypotheses

Our comparison experiments explore the following hypotheses:

Hypothesis 1: *Blended taint analysis can scale to real-world JavaScript programs.*

Hypothesis 2: *Blended taint analysis is more accurate than pure static taint analysis, capable of discovering more security violations and eliminating some false alarms.*

5.1.3 Benchmarks

The experiments were conducted with two sets of benchmarks. The first set of benchmarks was produced by an undergraduate researcher working as a web tester, who had no knowledge of blended taint analysis. She tested 12 websites that are among the top 25 most popular sites on *www.alexa.com* using modified *TracingSafari*. She was instructed to explore different functionalities on webpages. Statistics for these benchmarks are given in Table 1[4]. *Page count* is the number of webpages executed at each website and then analyzed. *Trace count* is the total number of page traces collected for each website.

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

The second set of benchmarks was a collection of JavaScript programs extracted from real websites consisting of *eval* invocations [10]. The JavaScript code in the second set of benchmarks was simplified for the purpose of *eval* transformation. Only a small portion of the real application was included and important pieces of code, (e.g., invocations from the *DOM* to event handlers), were removed. These benchmarks were only used to illustrate the capability of blended analysis in handling the *eval* construct shown in the case studies (Section 5.3).

We will be posting the traces we used for our blended analysis and code for our blended taint analysis of JavaScript online at our PROLANGS@VT website[5] by the end of February 2013.

5.2 Blended Taint Analysis Results

5.2.1 Analysis Time

Table 2 presents the time performance of the static phase of blended taint analysis, Static Taint$^+$, and Static Taint$^-$ each run under a limited time budget of 10 minutes suggested in [22].[6] Columns 2 and 5 present the number of webpages that could not be analyzed within 10 minutes by Static Taint$^+$ and Static Taint$^-$, respectively. Static Taint$^+$ only was able to fully analyze two websites, *linkedin.com* and *bing.com*. For some sites, Static Taint$^+$ did not scale on all/most webpages, (e.g., *yahoo.com* and *amazon.com*). Static Taint$^-$, which does not analyze library code, was capable of analyzing most JavaScript code from pages on the websites. To sum up, Static Taint$^+$ timed out on 108 out of 263 webpages and Static Taint$^-$ timed out on 12 out of 263 webpages. Blended taint analysis, on the other hand, was able to finish analyzing all selected page traces within the time limit.

Columns 4, 7 and 8 show the analysis time of each website averaged over those webpages that were not timed out for Static Taint$^+$, Static Taint$^-$ and blended taint, respectively. The time cost of the static phase of blended taint analysis is the total time of multiple static taint algorithm applications to each trace on a webpage. In Table 2, the average page analysis time of blended analysis exceeds that of Static Taint$^-$ on all websites but *google.com*. This is mainly caused by the fact that Static Taint$^-$ only analyzes JavaScript application code.

In [24], we ran both our modified instrumented Safari and an original Safari on JSBench [17], a JavaScript benchmark generated from real websites, to observe the instrumentation overhead. 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.

In conclusion, Static Taint$^+$, analyzing code including large libraries (e.g., *jquery*), could not complete analysis of 41% of the webpages we examined. Static Taint$^-$, only analyzing application code, could not complete analysis on 4.6% of the webpages. Blended analysis, on the other hand, did not use some of the costly models applied in the *WALA* static infrastructure, (e.g., *apply* and *call*) and ran to completion on all websites, demonstrating the focusing power of dynamic information and pruning. Thus, if we assume that an efficient instrumented version of a browser can be constructed, then given our timings for these analyses, we have support for **Hypothesis 1**.

5.2.2 Taint Analysis Results

Table 3 shows the results of the blended and the pure static taint analyses. Six of the 12 websites we tested contained security vulnerabilities. We report the number of unique alarms for each website. Duplicate alarms originated from the same cloned JavaScript code contained in different webpages. Each alarm was checked manually to determine if it was a *true positive* (i.e., there actually exists at least one flow from

[4]There are webpages cannot be parsed by *WALA*. Table 1 lists the benchmarks used in the experiment that are analyzable by *WALA*.

[5]http://prolangs.cs.vt.edu/

[6]We extended the static analyses time limits to 15 minutes with few observed differences in solutions. New work [6] presented an unsafe static analysis that improved performance.

Table 2: Taint analysis time

Website	Static Taint$^+$			Static Taint$^-$			Static phase of blended taint
	# pages timed out	# pages analyzed	Average time (sec.)	# pages timed out	# pages analyzed	Average time (sec.)	Average time (sec.)
facebook.com	14	13	28.5	0	27	19.2	29.4
google.com	13	9	39.2	1	21	22.4	14.2
youtube.com	10	5	57.3	2	13	13.9	37.4
yahoo.com	24	6	33.0	3	27	12.1	48.3
wikipedia.org	2	25	18.1	2	25	18.1	23.0
amazon.com	9	0	-	0	9	7.7	32.9
twitter.com	5	27	42.8	1	31	14.0	62.3
blogspot.com	6	3	27.3	0	9	14.8	18.8
linkedin.com	0	32	28.8	0	32	21.7	39.4
msn.com	10	3	38.0	1	12	25.3	42.4
ebay.com	15	25	21.1	2	38	12.7	18.5
bing.com	0	7	16.5	0	7	16.5	27.4

Table 3: Taint analysis results (Results in Static Taint$^-$ columns marked * mean the same alarms were reported by Static Taint$^+$)

Website	Static Taint$^+$		Static Taint$^-$		Blended Taint	
	true positive	false positive	true positive	false positive	true positive	false positive
youtube.com	1	1	1	-	4	-
twitter.com	1	-	1*	-	3	-
linkedin.com	1	1	1*	1*	1	1
msn.com	-	-	-	-	2	-
ebay.com	2	-	-	-	3	-
bing.com	-	1	-	1*	-	-
totals	**5**	**3**	**3**	**2**	**13**	**1**

a source to a sink) or not (i.e., false alarm or false positive). For the sink-source pairs that flowed into an *eval* invocation reported for Static Taint$^+$ or Static Taint$^-$, we manually checked if there actually was a taint violation. Blended taint analysis reported 14 unique source-sink pairs from the 6 websites; only one of them was a false positive. Static Taint$^+$ reported 8 *unique* source-sink pairs from 5 websites; 3 of them were false positives. Static Taint$^-$ reported 5 *unique* source-sink pairs from 4 websites; 2 of them were false positives.

Although Static Taint$^+$ timed out on many webpages, it was able to discover 5 true positives, 3 of which were not discovered by Static Taint$^-$. This suggests that JavaScript library code can be involved in a source to sink flow so that the libraries should be analyzed or modeled as precisely as possible. Static Taint$^-$ was able to locate only one different true positive from Static Taint$^+$ on *www.youtube.com*, although it analyzed many more webpages.

The blended taint analysis results in columns 6 and 7 in Table 3 support **Hypothesis 2**:

- Blended taint analysis discovered all the true positives that Static Taint$^+$ or Static Taint$^-$ reported.
- Blended taint analysis found 8 additional true positives that Static Taint$^+$ did not report and 10 additional true positives that Static Taint$^-$ did not report.
- For the 7 true positives detected by blended analysis, but not by either static analysis, 4 of them came from dynamic constructs the static analyses could not handle, while 3 of them may have been found statically with more time.
- Blended analysis eliminated 2 false positives reported by Static Taint$^+$ and 1 false positive reported by Static Taint$^-$, leaving only one false positive reported in common with the two static analyses.

5.3 Case Studies

In this section, we use some of the taint analysis results from section 5.2.2 as examples to illustrate the benefits of blended analysis. We also test the capability of blended analysis of monitoring *eval* on the second set of benchmarks to compare with static *eval* transformations [10].

5.3.1 Static Analysis False Positive from bing.com

In the JavaScript source from *www.bing.com*, there is an *eval* invocation that simply evaluates the value of a property of some object:

$$eval(n.responseText)$$

Both Static Taint$^+$ and Static Taint$^-$ report an alarm from an user input to this *eval* site through the variable n. We found n is an alias of the $XMLHttpRequest$ object whose fields are sensitive. The input variable did flow into the $XMLHttpRequest$ object through other properties so that n is tainted. However, the program does not contain an integrity violation if the sensitive methods of $XMLHttpRequest$ object are not called. Pure static taint analyses are incapable of knowing that the code in *eval* does not flow into any of the sensitive fields (i.e., sinks) so that the report is actually a false positive. Blended taint analysis observed the code generated from the *eval* of $n.responseText$ is not sensitive; hence, no security violation was reported.

5.3.2 Static Analysis False Negatives

There are several cases where blended taint analysis discovered true positives while the pure static analyses did not. Because blended taint analysis is more scalable than Static Taint$^+$, analyzes much more JavaScript code than Static Taint$^-$, and captures dynamic constructs that both pure static analyses do not, it produces more solutions from the code not analyzed by the static analyses. An interesting case from *www.ebay.com* demonstrates the origin of a static analysis false negative. We observed that when *ebay.com* was executed, a third party JavaScript code from the domain *bluekai.com* was loaded. It is usual for websites to include JavaScript code from other places; however, this code exhibits malicious behavior.

```
setTimeout("bkObj.clearDiv('bk_pl_520')",1001);

var bkObj = {
   clearDiv: function(divId) {
      var divRef = document.getElementById(divId);
      divRef.innerHTML=''; }
   };
}
```

Figure 8: Taint violation example from *ebay.com*

In JavaScript, $window.setTimeout$ and $window.setInterval$ functions are other ways to include strings that will generate dynamic code. The executable code

$$bkObj.clearDiv"('bk_p l_5 20')"$$

was observed by blended taint analysis and should be treated as a source because it comes from untrusted code. The tainted variable $divRef$ flows into a sink that rewrites the $innerHTML$ property of a DOM element. Static Taint$^+$ and Static Taint$^-$ did not report this true positive because the tainted source was called by run-time generated code which was not analyzed.

5.3.3 Blended vs. Static eval Transformation

Jensen *et al.* developed a static approach that was capable of transforming some *eval* calls into other language constructs to expand the applicability of static analysis [10]. We tested on the second set of benchmarks to check if blended analysis was able to monitor *eval* calls that were not successfully analyzed in [10].

For the example in Figure 3, blended analysis observed 19 calls to $eval(to)$ with the actual code recorded as

$$typeof(s.propX)$$

where X is substituted by each number between 1 and 19. Blended analysis also observed 2 calls to $eval(xe)$ and $eval(ex)$ with the actual code

$$s.propY = myUe(s.propY)$$

and

$$s.eVarY = myCp(s.propY,' D = cY')$$

respectively where Y is either 1 or 2. This collected code was represented as three call graph nodes in the dynamic call graph (each node contained the set of code observed for each *eval* call site). These nodes then could be analyzed by a blended analysis.

5.4 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) Because the executions of websites were collected by one undergraduate student, we may have introduced a bias in terms of the pages explored. To avoid this as much as possible, the undergraduate researcher collected executions without knowledge of the JavaScript website code or our analyses. (iii) Although we used websites listed at *alexa* as most popular, we cannot know how representative our input is of normal website usage. (iv) Although these initial experiment results are promising, more empirical investigation will be necessary for a stronger validation of our hypotheses.

6. 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 and analyses of some dynamic language features of JavaScript; (iii) security analyses for JavaScript.

Blended analysis of Java. Blended analyses for Java [4, 5] 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 appropriate executions for a client problem. The complexity of dynamic analysis for JavaScript far exceeds that of 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*). Blended analysis for both Java and JavaScript use calls and object creations observed for pruning unexecuted code; for JavaScript the number of arguments of a variadic function is also used for 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.

Studies and analyses of dynamic features. 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 [19]. The behaviors studied include call site dynamism, function variadicity, prototype-chain property changes, 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. Ratanaworabhan, *et al.* also presented a related study on comparing the behavior of JavaScript benchmarks, (e.g., *SunSpider* and *V8*), with real Web applications [16]. Their results showed numerous differences in program size, complexity and behavior which suggest that these benchmarks are not representative of JavaScript usage. This study motivated us to evaluate our blended analysis on website codes.

Studies of *eval* use in JavaScript applications [18] show that the *eval* construct, which can generate code at runtime, is widely used; therefore, we focussed on *eval* in blended analysis. Based on these observations, Meawad, *et al.* [13] presented *Evalorizer* to remove *eval* from real-world website codes. The approach matches the *eval* calls to patterns that can be transformed to easier-to-analyze JavaScript idioms. The removal of *eval* is semi-automated because it requires programmers to validate the transformed code. Jensen, *et al.* [10] 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. Both of these transformation techniques seek to enable static analysis of dynamic features, a goal shared by blended analysis. However, blended analysis expands the applicability of static analysis by recording the actual code generated in the *Dynamic Phase*.

Security analysis for JavaScript. Various analysis approaches have been applied to JavaScript security. Guarnieri, *et al.* [8] 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 Infrastructure* of blended taint analysis also uses the *WALA* infrastructure so that we share some models for JavaScript constructs in common with this work.

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 dynamic constructs 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 the more common dynamic features used by real websites.

Guha, *et al.* presented a static analysis to extract a model of expected client behavior to prevent attacks [9]. Their approach obtained dynamically loaded JavaScript code from developers, to avoid generating an incorrect or incomplete model of program behavior. Blended analysis automatically collects the dynamic code.

Maffeis, *et al.* [11] presented techniques to ensure JavaScript safety through *filters*, *rewriting* and *wrappers*. Filtering is a one-time static analysis to reject untrusted code if it does not conform to certain criteria. Rewriting inserts run-time checks to prevent undesirable actions. Wrapping uses run-time checks to ensure that the trusted environment is not used maliciously by untrusted code. This approach was studied on a subset of JavaScript, *Facebook FBJS*. Similarly, Yu, *et al.* [25] also implemented a set of JavaScript security policies through rewriting (i.e., run-time checking). These are dynamic online approaches to security checking. Blended taint analysis is an offline technique to identify data integrity violations.

Barth, *et al.* identified a class of vulnerabilities, cross-origin JavaScript capability leaks [1]. The points-to relation of the program is dynamically recorded *via.* an instrumented browser and then used to detect these vulnerabilities. The problem also can be cast as information flow. The approach collects all explicit and implicit pointers. Blended taint analysis differs from this approach because our lightweight *Execution Collector* only collects a limited set of operations to focus and refine the subsequent static analysis.

Saxena, *et al.* [20] applied a symbolic execution technique to find client-side code injection vulnerabilities. The approach is combined with automatic GUI exploration to build the end-to-end system, *Kudzu*. *Kudzu* takes the URL of a Web application as input and generates a high-coverage test suite. The test suite generated by this approach may be useable as input to *JSBAF*.

Chugh, *et al.* [2] presented an information flow analysis for JavaScript. The staged approach analyzes the statically visible code first and then incrementally analyzes the dynamically generated code. JavaScript blended analysis differs from this approach 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.

Tripp, *et al.* [23] combined black-box testing and static analysis to detect JavaScript security vulnerabilities. A web crawler was applied to retrieve webpages and the dynamically loaded/generated code was analyzed statically. Blended analysis provided more dynamic information (e.g., calls) to specialize the static phase. The web crawler can be used in the dynamic phase to ensure good coverage of the website.

Vogt, *et al.* presented a hybrid approach to prevent cross-site scripting [15]. In this work, dynamic taint analysis tracks data dependencies precisely and static analysis is triggered to track control dependencies if necessary. Blended analysis combines dynamic and static analyses differently.

7. CONCLUSION

Analyzing JavaScript programs statically is difficult because its dynamic features cannot be precisely modeled. *JSBAF* was designed to address this challenge. We instantiated *JSBAF* to perform blended taint analysis for JavaScript and experimented with our implementation on websites chosen from the top 25 reported by *alexa.com*. Comparison to two pure static taint analyses (i.e., Static Taint$^+$, Static Taint$^-$) showed that:

- while Static Taint$^+$ could not finish analysis of approximately 41% of the pages from these websites within 10 minutes, the static phase of our blended taint analysis ran to completion on all webpages. Static Taint$^-$ was more capable than Static Taint$^+$ in terms of the number of pages analyzed;

- blended taint analysis discovered 13 unique security violations on 6 of the 12 websites analyzed, reporting only 1 false alarm. In contrast Static Taint$^+$ found 5 violations and 3 false alarms and Static Taint$^-$ found 3 violations and 2 false alarms;

- case studies of actual JavaScript website code illustrated how blended taint analysis was more successful than either of the two static taint analyses.

Thus, blended taint analysis had significantly better performance and accuracy than the static techniques. These investigations attest to the promise of blended analysis of JavaScript for useful client problems.

In future work, we plan to explore more clients for JavaScript blended analysis, including aiding program understanding of evolving website code. We are interested in providing more precise models for prototype-chain property changes and using these results in analysis. We also want to investigate the applicability of our analysis technique for other dynamic programming languages.

8. ACKNOWLEDGEMENTS

We would like to thank Nasko Rountev and Frank Tip for their comments on earlier versions of the paper.

9. REFERENCES

[1] 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.

[2] 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.

[3] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[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. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for javascript ide services. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 752–761, 2013.

[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, 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.

[9] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for ajax intrusion detection. In *Proceedings of the 18th international conference on World Wide Web*, WWW '09, pages 561–570. ACM, 2009.

[10] 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.

[11] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, pages 505–522, Berlin, Heidelberg, 2009. Springer-Verlag.

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

[13] F. Meawad, G. Richards, F. Morandat, and J. Vitek. Eval begone!: semi-automated removal of eval from JavaScript programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 607–620, 2012.

[14] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, Jan. 2005.

[15] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528. IEEE Computer Society, 2010.

[21] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234.

[22] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of the 26th European conference on Object-Oriented Programming*, ECOOP'12, pages 435–458. Springer-Verlag, 2012.

[23] O. Tripp and O. Weisman. Hybrid analysis for JavaScript security assessment. ESEC/FSE, 2011.

[24] S. Wei and B. G. Ryder. A practical blended analysis for dynamic features in JavaScript. Technical Report TR-12-18, Virginia Tech, 2012.

[25] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 237–249, New York, NY, USA, 2007. ACM.