A Practical Blended Analysis for Dynamic Features in JavaScript

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

ABSTRACT

The JavaScript Blended Analysis Framework is designed to perform a general-purpose, practical combined static/dynamic analysis of JavaScript programs, while handling dynamic features such as run-time generated code and variadic functions. The idea of blended analysis is to focus static analysis on a dynamic calling structure collected at runtime in a lightweight manner, and to refine the static analysis using additional dynamic information. We perform blended points-to analysis of JavaScript with our framework and compare results with those computed by a pure static points-to analysis. Using JavaScript codes from actual webpages as benchmarks, we show that optimized blended 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.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.3.4 [**Languages**]: Processors

General Terms

Experimentation, Languages, Measurement

Keywords

Program analysis, points-to analysis, JavaScript

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[1] use JavaScript [11]. 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 and code injection [5, 11, 3, 12, 20, 2], and to improve performance through trace-based, just-in-time compilation techniques [8, 13]. 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 [21, 23, 22] 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*). Static reasoning about dynamically generated code is very difficult; the analysis has to be very conservative to remain *safe*, in a data-flow sense [19]. Richards *et.al* [22] 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* [23], 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. All of these dynamic features make it hard to precisely reason about JavaScript applications. Richards *et.al* [23] point out that existing work tends either to ignore or to make incorrect assumptions regarding the dynamism of JavaScript codes.

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

[1] http://www.alexa.com

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

its calling structure [6, 7]. 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.[2] Other researchers defined more precise string analyses to enable better modeling of some reflective calls in Java. Livshits *et al.* [17] designed a static analysis that determined the targets of reflective calls by tracing the flow of class name strings. In the presence of input-dependent targets, either user-provided information or approximations based on type casts were used. Christensen *et al.* [4] focused on a static string analysis that improved the accuracy of Java call graphs containing reflective calls using *Class.forName*. But these previous approaches are insufficient to deal completely with reflection in Java.

In contrast to these analyses, which captured or approximated reflective call targets, 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 wide-spread usage of these dynamic features in JavaScript applications, their capture is important, because a pure static analysis misses them (e.g., when an *eval* contains a JavaScript code string which contains user inputs) or approximates 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. As such, 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 that by judiciously combining dynamic and static analyses in an unsafe [19] analysis of JavaScript, we can account for the effects of dynamic features not seen by pure static analysis, while retaining sufficient accuracy to be useful. Intuitively, blended analysis focuses a static analysis on a dynamic calling structure collected at runtime, and further refines the static analysis using additional information collected by a lightweight dynamic analysis. For our framework, we analyze multiple executions of a JavaScript code with good program coverage, in order to obtain analysis results for the entire program.

We have instantiated our JavaScript Blended Analysis Framework to do points-to analysis, in order to show time cost and precision differences in empirical comparison to a pure static analysis. Points-to analysis of JavaScript is an important *enabling* analysis, supporting many static clients (e.g., side-effect and information-flow analyses). Although pure static points-to analyses for JavaScript [15, 9] have been presented by researchers focusing on specific language features, these analyses miss key dynamic features of the language.

The major contributions of this paper are:

JavaScript Blended Analysis Framework. Our framework is designed to better analyze the language's dynamism and to allow a general-purpose data-flow analysis as a module of the infrastructure. The blended algorithm is optimized by selection of a *cover set* from among the executions observed to reduce total analysis time. The data-flow safety of solutions obtained also is discussed.

[2]We use the term *pure static analysis* to refer to an analysis based on monotone data-flow frameworks [19].

```
1:  <script src="http://sample.com/myscript.js"></script>
2:  <script>
3:     function Sample() {
4:          if(arguments.length == 1) {
5:               var a = arguments[0];
6:               if(typeof (a) == "Array") {
7:                    arry =  new Array(a.length);
8:               } else m2(a);
9:          } else if(arguments.length < 5) {
10:              var b =[];
11:              for(c = 0; c < arguments.length-1; c++) {
12:                   b[c] = eval(arguments[c]);
13:                   m3(b[c]);
14:              }
15:          } else m4(arguments);
16:    }
17: </script>
```

Figure 1: JavaScript example

Implementation of a blended points-to analysis for JavaScript. The framework is instantiated for points-to analysis of 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. The framework is implemented using *TracingSafari* [23], and the *IBM T.J. Watson Libraries for Analysis* (*WALA*) open-source framework.[3]

Experimental evaluation. The empirical results of the blended points-to analysis have been compared with those from a pure static points-to analysis for JavaScript already available in *WALA*. The comparison shows that optimized blended analysis for JavaScript can obtain good coverage of the static analysis solution (i.e., 78.0%-93.0% of static analysis results on average per website) and finds additional (i.e., missed) points-to pairs resulting from dynamically generated/loaded code which comprise 1.4%-9.9% of the final solution on average per website, quite significant for some websites. The cost of the optimized blended analysis is shown to exceed the cost of static analysis by only 31.6% on average per website.

Overview. The rest of this paper is organized as follows: Section 2 uses an example to illustrate the dynamism of JavaScript. Sections 3 and 4 introduce the JavaScript Blended Analysis Framework instantiated for blended points-to analysis. Section 5 presents our experimental results. Section 6 discusses work related ours, and Section 7 offers conclusions and future work.

2. MOTIVATING EXAMPLE

In Figure 1 we present a sample JavaScript program containing dynamic characteristics to illustrate the challenges of analyzing dynamic languages. Similar coding style appears in real websites.

Line 1 in Figure 1 is a link to JavaScript source. A statement like this is widely used in websites to link external widgets or to provide additional interactions. This piece of JavaScript code may be loaded at runtime when some event is triggered. It is important to analyze this code because it can affect the behavior of the webpage. The rest of the example declares a JavaScript function. In line 3, the function

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

signature is given without any arity; however, it is designed to be called with different arguments. The function body illustrates that this function can behave with totally different functionality, depending on the number of arguments. Lines 5 to 8 execute if *arguments.length* is 1. The creation of *arry*(line 7) depends on the actual type of the argument *a* provided, which only can be decided at runtime. Lines 10 to 14 demonstrate another case when less than 5 arguments are provided. Line 12 uses the *eval* construct to evaluate the string associated with the value of $arguments[c]$. Whether or not this code is actually executed can be decided only at runtime since statically we cannot learn the value of $arguments[c]$. It is important to analyze this code because malware can use *eval* as an obfuscation mechanism; for example, the user input $arguments[c]$ can load malicious JavaScript code which then will be run by the *eval*.

This small example contains dynamically loaded code, variadic functions, the *eval* construct, and dynamic type dependent object creation, which all contribute to the dynamism of JavaScript. These features cannot be precisely modeled via pure static analysis. These constructs are the focus of our JavaScript Blended Analysis Framework.

3. FRAMEWORK OVERVIEW

In this section, we present an overview of the design of the JavaScript Blended Analysis Framework and explain the underlying new, general-purpose blended analysis for JavaScript. In Section 4 we give details of how we instantiated this framework for points-to analysis of JavaScript programs.

Our design goal for the JavaScript Blended Analysis Framework is to have 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 should 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. Figure 2 illustrates the three components of the framework: the *Dynamic Analyzer* collects run-time information about the applications; the *Selector* is an optional component designed to keep the analysis cost practical; the *Static Analyzer* analyzes the program representation that incorporates information obtained from the *Dynamic Analyzer*.

In our framework, JavaScript applications are instrumented and dynamic analysis produces run-time information for each execution. This information is used to construct a dynamic calling structure for the execution so that inter-procedural static analysis can use it as a program representation for analysis. In addition, other dynamic information that helps to improve static analysis accuracy can also be collected. As shown in Figure 2, the *Dynamic Analyzer* outputs data for multiple executions.

Optional use of the *Selector* may reduce the number of executions used as inputs to the *Static Analyzer*. Although each execution represents a different path in a JavaScript application, multiple executions can contain some of the same methods of the program. The *Selector* chooses executions that may have significant positive impact on the results of JavaScript blended analysis; by using fewer than all executions collected, the *Selector* lowers the overall analysis cost, while maximizing code coverage.

The *Static Analyzer* consists of a pure static analysis algorithm augmented by dynamic information about the executions as well as the application source code for the methods executed as input. Because our JavaScript blended analysis aims to handle the dynamism of the language, this may include dynamic information that would be very hard (or impossible) for a pure static analysis to approximate well. The *Static Analyzer* analyzes each execution separately and then combines the results of these analyses into a solution.

Two goals of our research are to study how well the new JavaScript Blended Analysis Framework can analyze programs written in a heavily dynamic, late binding language, and to better understand the tradeoffs between safety and practicality represented by blended analysis.

4. BLENDED POINTS-TO ANALYSIS

This section describes our instantiation of the JavaScript Blended Analysis Framework for points-to analysis. We present both the design and implementation details for each component in Figure 2. Each component of the framework is designed to handle some of the challenges of analyzing JavaScript applications. We also discuss analysis safety.

4.1 Dynamic Analyzer

Our focus is on JavaScript code found on many webpages. We will refer to the JavaScript code on a single webpage as a JavaScript program. Our framework is targeted to analyze such JavaScript programs which have been shown to have attributes different from some standard JavaScript benchmarks such as *SunSpider*[4] and *V8*[5] [23, 21].

We believe that an instrumented web browser is capable of capturing the dynamic executions needed by blended analysis through daily usage, without affecting website performance. For a practical analysis, we require that the *Dynamic Analyzer* be lightweight, although it has to collect sufficient run-time information to reason about the dynamic constructs captured in support of the subsequent static analysis.

Our *Dynamic Analyzer* instrumentation infrastructure relies on a specialized version of the WebKit module of Apple's Safari web browser. *TracingSafari*, an instrumented version of WebKit[6] JavaScript engine, was developed for characterizing the dynamic behavior of JavaScript programs [23]. 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 very lightweight, we modified *TracingSafari* to collect only the information required by blended analysis.

In blended analysis, the *Dynamic Analyzer* builds a graph representation of a JavaScript program. An execution can be precisely represented by recording all the instructions; however, that approach is too costly. Dynamic analysis can provide the exact call tree of each execution by collecting all the function calls at runtime. For each execution, we constructed a call graph that contains the functions actually executed.

Because JavaScript is a dynamically typed programming

[4]http://www.webkit.org/perf/sunspider/sunspider.html
[5]http://v8.googlecode.com/svn/data/benchmarks/v6/run.html
[6]webkit.org

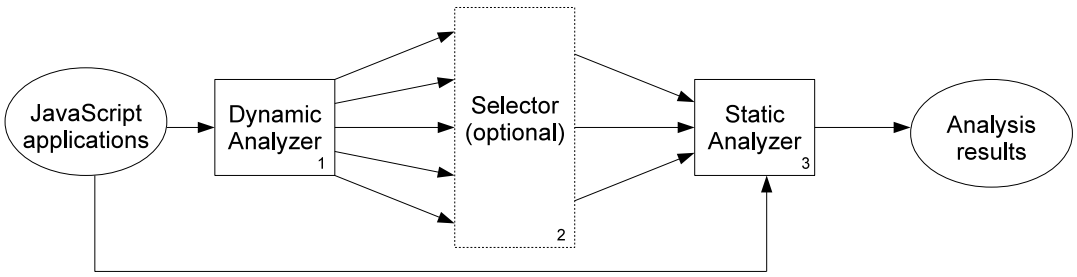

Figure 2: JavaScript Blended Analysis Framework

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 5-7 in Figure 1. In blended analysis, our *Dynamic Analyzer* 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 interprocedurally and intra-procedurally.

Pure static analysis of JavaScript can extract JavaScript source code from webpages. During execution, however, invocations to reflective constructs such as *eval* may generate new JavaScript code. Recall that this generated code may be difficult to model statically because the evaluated string parameter of *eval* may contain variables whose values are not knowable until runtime. In addition, webpages often download JavaScript codes and load them dynamically. Pure static analysis has no access to this downloaded code whereas our *Dynamic Analyzer* is able to capture all the source code file loads, capturing all code generated or loaded. This dynamic code is traced in the same manner as the other parts of the program, with instrumentation of both functions and object allocation sites.

Function variadicity is another dynamic feature of the JavaScript language. A function can be called with an arbitrary number of arguments, regardless of the function 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 $arguments$ variable associated with each function. Sometimes, the actual behavior within a function can be differentiated by its number of arguments as in the example 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 *Dynamic Analyzer* captures the actual number of arguments for each call site and builds separate nodes in the call graph for instances of the same signature function with different numbers of arguments.

In summary, the *Dynamic Analyzer* used for blended points-to analysis instruments function calls (and captures their number of arguments), object allocations, and dynamically generated/loaded source code. This results in a relatively lightweight dynamic analysis.

4.2 Selector

The combined cost of the *Dynamic Analyzer* and the *Static Analyzer* contributes to the overhead of blended analysis over a pure static analysis. Since we use a lightweight dynamic analysis, the choice of static analysis algorithm to use on each of the multiple executions dominates the overall analysis performance. In this section, we present an optional optimization which selects a subset of observed executions to serve as input to the *Static Analyzer*.

Our JavaScript Blended Analysis Framework requires multiple executions because we would like to analyze as much of the JavaScript program as possible, certainly more than can be explored by just a single execution. However, since a static analysis must be performed on each execution, using more executions increases the cost of blended analysis. Multiple executions may overlap in the methods they execute, because different executions can run over the same areas of the program. The *Selector* tries to minimize the number of executions used by the analysis to cover the JavaScript program while covering as much program behavior as possible. We hypothesize that there is a subset of the executions 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 executions that is the same as or *close to* the solution obtained using *all* executions, while reducing the overall overhead of blended analysis.

Figure 3 presents the execution selection algorithm. The set of all the collected executions from the *Dynamic Analyzer* is input to this selection algorithm. The heuristic used in function *dist* in line 8 will be explained below. It calculates a linear combination of factors which emphasize using executions that cover more methods, explore more different object types and contain dynamically generated/loaded code. The threshold T, a value between 0 and 1, is an input to the algorithm that can be adjusted by the user of the framework. The threshold maintains the balance between blended analysis performance (i.e., fewer executions) and accuracy (i.e., more executions). The set of executions selected is an input to the *Static Analyzer*.

The algorithm in Figure 3 works as follows. At line 1, an execution e_i is randomly chosen to be the first selected from all the executions. Line 2 initializes the variable $Dist$ used as a criterion to select executions. During the algorithm, $Dist$ is a score of what a particular execution will add to those executions already in the selected set Ex_{best}. Lines 3-13 describe the selection process which continues until the threshold T is reached or there are no longer any executions to add. Lines 4 and 5 process the sets by removing the selected execution from Ex and adding it to Ex_{best}. Lines 7 to 12 comprise a loop to select the best next candidate execution to add. The *dist* function compares a candidate execution with those in the already selected set Ex_{best} to calculate the corresponding $Dist$ value. The execution with the largest $Dist$ value is chosen as the next candidate to

Input: Ex(All executions collected by dynamic analysis); T(Threshold of the selection algorithm)
Output: Ex_{best}(Executions selected to do static analysis)

```
1: e_i ← random(Ex)
2: Dist ← T
3: while Dist ≥ T and Ex is not empty do
4:    Ex − {e_i}
5:    Ex_best ← Ex_best ⋃ {e_i}
6:    Dist ← −1
7:    for each e in Ex do
8:       if dist(Ex_best, e) > Dist then
9:          Dist ← dist(Ex_best, e)
10:         e_i ← e
11:      end if
12:   end for
13: end while
```

Figure 3: Selector algorithm

add.

Heuristic. To explain the heuristic used by the *dist* function, we need to explain three factors: method coverage, created object type coverage and dynamically generated/loaded code coverage. For each execution e there is a set of associated functions M_e and dynamically generated/loaded code C_e. Within each method $m_i(e) \in M_e$, there is a set of observed allocated types $T_{m_i(e)}$. The set of executions S can be represented by an aggregated set of methods M_S and dynamically generated/loaded code C_S; method $m_i(S) \in M_S$ contains a set of those types $T_{m_i(S)}$ allocated on some execution in S.

During the selection process, we need to ensure that most observed functions and object creation sites are covered by the selected execution set, in order to get the best points-to information possible. If a function is not analyzed, we cannot obtain any point-to pairs resulting from its execution. Because our analysis framework is particularly aimed at dealing with dynamically loaded code, we wish to select executions with this property with a high priority.

Function coverage. With a set of selected executions S, we would like the next execution to be chosen to add a maximal number of new functions. The number of functions in both S and e is $|M_S \cap M_e|$, so that $|M_S \cup M_e| - |M_S \cap M_e|$ calculates the number of new functions that will be added to M_S, if e is added to S. We normalize this measurement thusly:

$$dist_{func} = \frac{|M_S \cup M_e| - |M_S \cap M_e|}{|M_S \cup M_e|}$$

Object type coverage. We observe that if different types of objects are allocated in the same function on different executions, then (i.) different intra-procedural paths may be executed, and/or (ii.) there may be invocations with different targets that result in different inter-procedural paths. So, if additional object types would result from adding e to S, then the *Selector* may explore more program paths. Since all object allocation information is associated with the corresponding function in which the creation takes place, we measure the difference in number of object types in the context of each function. The set of functions occurring both in S and e is $M_{S\cap e} = M_S \cap M_e$. For each such function $m_i \in M_{S\cap e}$, the sets of types created in S and e are $T_{m_i(S)}$ and $T_{m_i(e)}$, respectively. We calculate the number of additional types added if e is added to S as:

$$dist_{T_{m_i}} = \left|T_{m_i(S)} \cup T_{m_i(e)}\right| - \left|T_{m_i(S)} \cap T_{m_i(e)}\right|$$

The aggregate difference of object type allocations between S and e is measured through the aggregation of all methods $M_{S\cap e}$ in common

$$dist_{obj} = \frac{\sum_{i=1}^{|M_{S\cap e}|} dist_{T_{m_i}}}{\sum_{i=1}^{|M_{S\cap e}|} \left|T_{S(m_i)} \cup T_{e(m_i)}\right|}$$

Dynamic code generation/loading. To find additional points-to pairs due to dynamically generated/loaded code, the framework must analyze executions on which this occurs. If the dynamic code loaded during execution e, C_e, is different from the code loaded in S, C_S, then we set $dist_{dyn} = 1$; otherwise, $dist_{dyn} = 0$.

Combining factors. The *Selector* considers all three factors when deciding the next execution to add. We hypothesize that dynamic code produces some points-to results that pure static analysis cannot obtain and that function coverage affects program coverage more than the object types allocated. In our current implementation, the weights for dynamic code, function, and object allocations are 0.5, 0.4, and 0.1, respectively. The $dist(S_{ex}, e)$ function returns the value of:

$$0.5 \times dist_{dyn} + 0.4 \times dist_{func} + 0.1 \times dist_{obj}$$

These weights can be adjusted based on analysis requirements.

4.3 Static Analyzer

The *Static Analyzer* of our JavaScript Blended Analysis Framework applies a pure static algorithm to each execution, and then combines the results of multiple executions into the final blended analysis solution. The *Static Analyzer* takes as input the compile-time visible JavaScript source code, dynamic call structure, and dynamically created/loaded code recorded by an execution. The source code for the functions observed during execution is contained either in the compile-time available source or in the recorded dynamically generated/loaded code. The *Static Analyzer* also prunes function bodies based on the dynamic information gathered, in order to improve the analysis precision and performance.

Recall that we built our static infrastructure for analyzing JavaScript on the *IBM T.J. Watson Libraries for Analysis (WALA)* open-source framework.[7] *WALA* is a static analysis framework for Java that includes a JavaScript front-end. *WALA* parses JavaScript source code from a web-page producing an abstract syntax tree (AST), and translates the AST into *WALA* intermediate form. Several challenges of analyzing JavaScript, including prototype-chain property lookups and reflective property accesses, are addressed in *WALA*; however, *WALA* does not model reflective calls such as *eval* or *with* constructs [11]. *WALA* provides an Andersen-style flow- and context-insensitive points-to analysis for JavaScript which performs on-the-fly call graph construction.

In order to implement blended points-to analysis based on the static points-to algorithm in *WALA*, we modified the *WALA* implementation by removing the call graph construction and substituted use of our dynamic call graph representation derived from run-time information. Thus, in blended analysis the call graph is built from dynamic information and then input to subsequent static analysis, while in pure static

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

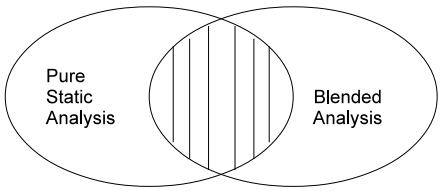

Figure 4: Relation between solutions obtained through pure static analysis vs. blended analysis

analysis, call graph construction interleaves with points-to analysis. Nodes in the dynamic call graph can originate from the dynamically generated code by an *eval* construct or JavaScript code downloaded during the execution. Since *WALA* does not model *eval* and static analysis does not extract code dynamically generated/loaded, the source code of some executed functions will not be available to *WALA*.

In addition, we had to extend the *WALA* intermediate representation in order to better deal with variadic functions in the call graph. In the *Dynamic Analyzer* for each function we record its actual number of arguments, so that the same function signature called with different numbers of arguments will result in multiple nodes in the dynamic call graph. In *WALA*, JavaScript functions are identified through source code declarations so that variadic functions cannot be distinguished statically. In our implementation, the *WALA* call graph representation 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. Pruning is an optimization technique applied in our blended analysis for Java. It was very effective removing approximately 30% of basic blocks from Java functions [7]. JavaScript blended analysis applies this same pruning technique and also prunes based on additional dynamic information, namely $arguments.length$, to provide an accurate control flow graph for a variadic function. Sometimes branches of variadic functions are determined by the value of $arguments.length$ (see example in Figure 1). In this case, that value can be used to prune the blocks on unexecuted branches. Thus, with pruning we are able to provide a more accurate approximation of the code within a variadic function. A future goal of our research is to study the effect of pruning on analysis of JavaScript, especially to see if precision is increased for such variadic functions.

4.4 Blended Analysis Safety

A pure static algorithm conservatively approximates an input JavaScript program that is constructed from the subset of the JavaScript language modeled. In blended analysis, the program input to the *Static Analyzer* is a conservative approximation of the actually executed JavaScript code, including source code for the executed methods observed. Intuitively speaking, blended analysis reasons only about those program executions being statically analyzed, rather than reasoning over all possible executions, producing a *safe* dataflow solution [19] for those executions. The term safe is used here to mean no false negatives for the part of the program covered by these executions. In this section we discuss the safety of blended analysis for JavaScript and contrast the program representations used and solutions found by pure static and blended analyses.

Analysis safety. Pure static analysis of JavaScript uses a call graph G_{sta}, a representation of a set of functions F_{sta} and their possible calling relations recognized from the statically accessible source code. The call graph, G_{sta}, representing a set of possible execution trees, is a conservative approximation of possible calling relations between the F_{sta} in the program. Recall that pure static analysis can not always model dynamic JavaScript constructs such as *eval* because there might be variables in the evaluated string expression whose value cannot be decided until runtime, such as the variable $arguments[c]$ in Figure 1, line 12.

Blended analysis, on the other hand, captures profiling information needed to model dynamic features, but may not be able to explore all executable paths in a JavaScript program. JavaScript blended analysis of an execution e analyzes a call graph representation of the set of observed functions, F_e. F_e consists of two subsets: (i.) $F_{e(sta)}$, a set of statically visible functions, $F_{e(sta)} \subseteq F_{sta}$, and (ii.) $F_{e(dyn)}$, a set of functions profiled in dynamically generated/loaded code during execution of e. The output of the *Dynamic Analyzer* is a call graph representation of e, G_e, which is then input to the *Static Analyzer*. Therefore, G_e contains all the inter-procedural invocations that occur when e is executed. G_e also is a conservative approximation of possible calling relations between the $F_{e(sta)}$ and $F_{e(dyn)}$ in the program.

Both G_{sta} and G_e may possibly introduce unexecuted inter- and intra-procedural paths in the JavaScript program. Therefore, both blended and pure static analysis construct conservative approximations of the possible program calling structure they analyze, (i.e., G_{sta} and G_e). Note that G_e is not necessarily a subgraph of G_{sta} because of the nodes and edges introduced by $F_{e(dyn)}$. Also, blended analysis may not execute all functions in F_{sta}.

Static analysis is a component of both pure static and blended analysis; in both cases, the static analysis produces a safe solution on the corresponding program representation, with no false negatives. The pure static analysis is safe with respect to the statically accessible JavaScript code. The blended analysis is safe with respect to the code corresponding to the observed execution. In some sense, blended analysis treats an execution as an entire program, building a conservative approximation of the calling structure and applying a safe static analysis to this representation. Therefore, blended analysis of a single execution is safe, in the data-flow sense of having no false negatives for the portion of the program it analyzes; in other words, blended analysis does not miss any true positives in the solution which occur due to the semantic effects of code in the execution being analyzed. Since our blended analysis unions the solutions on several executions to form the entire analysis solution, and each component solution is safe, therefore the entire solution is safe.[8]

The above argument ignores the pruning performed in blended analysis. The *Static Analyzer* analyzes the pruned program so that it is important to ensure that the pruned functions still conservatively approximate the sequence of instructions executed by e. If a function call site or object allocation site is not executed, we know this from the profiling by the *Dynamic Analyzer*. We prune all the basic blocks that contain unexecuted call sites and unexecuted object allocation sites; the statements that are post-dominated by these basic blocks are pruned as well. We also prune the branches

[8]Note: this argument holds whether all executions are used or a selection of executions is used.

that are not executed in variadic functions by knowing the value of *arguments.length*. Given these rules, it is obvious that we only prune code that is not executed, so that the code left after pruning is a conservative approximation of all the instructions executed on e. Therefore, the arguments given above hold, and the blended analysis solution is safe.

Comparison of solutions. There is a complicated relationship between the blended analysis solution and the pure static analysis solution for a particular program. The diagram in Figure 4 reflects the relationship between a pure static analysis and a blended analysis for JavaScript. The ellipses represent the solutions obtained for each analysis. Their intersection includes the part of the solution due to code within the static analysis program model and within those executions observed by blended analysis. In G_e, $G_{e(sta)}$ represents the statically discernible code executed by e and $G_{e(dyn)}$ represents the dynamic code executed by e. $G_{e(sta)} \subseteq G_{sta}$ because $E_{e(sta)} \subseteq E_{sta}$ and $V_{e(sta)} \subseteq V_{sta}$. Therefore, there may be parts of the blended analysis solution that also are contained in a pure static analysis solution.

The left part of the diagram shows that there may be a set of results that blended analysis does not calculate, which are covered by pure static analysis. This occurs when there is an unexecuted part of the program that contributes to the analysis solution (i.e., from $G_{sta} - \cup_{e_i} G_{e_i(sta)}$). Clearly, it may be difficult for blended analysis to explore sufficiently many executable paths to prevent this from happening. However, pure static analysis may introduce false positives by over-approximating the program and including results due to infeasible paths. These false positives may be (or may not be) avoided by blended analysis by its exclusive use of inter-procedural paths actually executed and by avoiding some unexecuted intra-procedural paths (as we explained in Section 4.3).

The right part of the diagram shows that the blended analysis solution may obtain results that a pure static analysis may not produce, because they correspond to program constructs difficult (or impossible) for static analysis to model due to the dynamism of JavaScript. For example, constructs such as an *eval* whose argument is not a string constant requiring run-time interpretation or JavaScript program segments loaded during execution (e.g., through an interpreted *url*) can hardly be modeled well by static analysis. $F_{e(dyn)}$ is the part of code on execution e unavailable to pure static analysis, so it cannot be analyzed. Blended analysis uses $G_{e(dyn)}$ as part of its call graph; thus, analysis of $G_{e(dyn)}$ may produce additional results not found by static analysis, as shown in the diagram.

5. EVALUATION

5.1 Experiment Design

Our blended points-to analysis for JavaScript has been tested with JavaScript code from eight of the most popular web sites in *Alexa* (www.alexa.com). The author who manually performed these instrumented explorations of websites in order to generate input for the analysis, had no knowledge of the underlying JavaScript code at the website, and tried to explore as much of the functionality of each webpage as possible. Each sequence of JavaScript instructions so obtained was then decomposed into a set of consecutive sequences of instructions from the same webpage called *traces*.[9] One input-generating session might explore several webpages, but it results in at least one trace for each page executed. Traces for the same webpage are inputs to the *Selector* and the *Static Analyzer*.

In the experiment, we compare blended points-to analysis with a pure static points-to analysis. Because blended analysis and pure static analysis treat each webpage as a separate JavaScript program, we are able to compare their points-to solutions. Our goal is to empirically evaluate blended points-to analysis for JavaScript in terms of its accuracy and performance.

Pure static analysis produces points-to results for the statically accessible program. The results contain actual points-to pairs (true positives) and false positives. In our experiment, we measure the difference between the points-to pairs produced by blended analysis versus those produced by a pure static analysis. With the goal of obtaining good coverage of the statically accessible program by our executions, we hope to achieve good coverage of the true positives in the pure static solution.

Points-to information due to dynamically generated/loaded code can only be found by blended analysis. We measure the number of extra points-to pairs produced by blended analysis to show the effect of dynamic code on analysis results. Since blended analysis applies a more accurate model for variadic functions through pruning, we would like to explore the analysis accuracy in future work.

Blended analysis is practical if the *Dynamic Analyzer* is lightweight and the *Static Analyzer* has acceptable overhead. The overhead of the original blended analysis consists of dynamic analysis and static analysis. Optimized blended analysis reduces the overhead of *Static Analyzer* and adds the extra overhead of *Selector*. We compare the performance of pure static analysis with optimized blended analysis to show that blended analysis has practical performance.

The experimental results are obtained on a 2.53 GHz Intel Core 2 Duo machine with 4 GB memory running the Mac OS X 10.5 operating system. The dynamic statistics for the input are given in Table 1.

Here *page count* is the number of pages executed at each website and then analyzed. Of course, it is not feasible to exhaustively explore all pages manually. *Trace count* is the total number of traces collected for each website. Note the number of traces on a webpage determines the number of separate static analyses required by our unoptimized blended analysis for JavaScript. The last two columns in Table 1 show the number of pages on which we observed *eval* calls and variadic functions, respectively.

5.2 Experimental Results

Algorithm Comparison. Table 2 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. The columns labelled *blended coverage* present the blended solution coverage of the static analysis solution as a percentage. More formally, assume there are n pages explored in a website. For each page i, the static analysis solution points-to pairs comprise set S_i and the blended analysis solution points-to pairs are set B_i. Then the percentages shown in columns 2

[9]The term *execution* used in the description of blended analysis in Section 2 thus refers to a *trace* in our experiments. A webpage is analyzed on the basis of several traces.

Website	Abbr.	Page count	Trace count	*eval* page	variadic functions
google.com	GOGL	203	2104	52	177
facebook.com	FBOK	138	1098	23	65
youtube.com	YTUB	122	579	19	29
yahoo.com	YHOO	52	265	21	13
baidu.com	BAID	49	147	6	16
wikipedia.org	WIKI	67	130	0	3
live.com	LIVE	54	226	10	44
blogger.com	BLOG	24	146	6	7
totals		709	4695	137	354

Table 1: 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.

Site	Blended coverage(%) of static results			Additional blended(%) results		
	original		optimized	original		optimized
	average	min. page	average	average	max. page	average
GOGL	90.5	15.1	89.7	**5.9**	60.3	**5.9**
FBOK	88.9	11.0	85.3	7.7	30.2	7.5
YTUB	89.4	20.9	89.1	**9.9**	52.0	**9.9**
YHOO	80.8	23.5	78.0	10.3	42.7	9.8
BADU	93.5	53.4	93.0	**6.7**	23.7	**6.7**
WIKI	92.2	70.2	92.1	-	-	-
LIVE	84.1	30.8	81.8	8.5	38.6	7.5
BLOG	85.2	34.3	83.8	**1.4**	2.3	**1.4**
geom. mean	88.1	32.4	86.6	7.2	35.7	7.0

Table 2: Original and optimized blended points-to analysis results for JavaScript

and 4 in table 2 are

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

The coverage of the static solution achieved by the original blended analysis (column 2) varies from 80.8% to 93.5%, with a range of standard deviations from 10.1% (*wikipedia.org*) to 21.5% (*blogger.com*). The overall average (i.e., 88.1%) indicates that a large number of the programs are modeled well by our blended analyses. For 64 of the 709 webpages analyzed, blended analysis obtained 100% of the static points-to solution. For 37.3% of the pages, blended analysis covered at least 95% of the static points-to solution. Nevertheless, for some individual webpages, the coverage was low. Column 3 shows the lowest coverage of a webpage in each of these websites. For example, for in one page of *facebook.com*, blended analysis produced only 11% of the points-to pairs found by static analysis. The low coverage was caused by low method coverage. We only explored two traces of this webpage and the method coverage was 21.2% while the average method coverage over the 138 *facebook.com* pages was 92.5%.

Column 4 presents a comparison of the optimized blended analysis points-to solution with the static analysis solution. Recall that the goal of the optimization is to reduce the overhead of blended analysis, while preserving as much of the points-to solution as possible (see Section 4.2). For the experimental results given in this section we used an optimization threshold of 0.02. The coverage results suggest that the optimization is effective in that little reduction in coverage is observed. On average for each website, the optimized blended analysis covered 1.6% fewer points-to pairs than the original blended analysis.

Effects of Dynamically Loaded Code. Columns 5 to 7 in Table 2 illustrate the effect of dynamically loaded code on the points-to solution. The data in columns 5 and 7 present 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 are 137 pages (19% of total pages) containing dynamically generated code by *eval* and 383 pages (more than 50%) containing new JavaScript code dynamically loaded at runtime.[10]

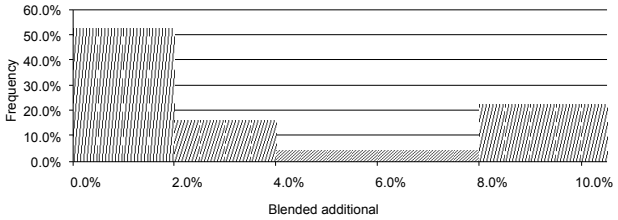

Figure 5: blended additional histogram

[10]We observed that often for initialization the same piece of dynamic 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. This website behavior was observed only for *facebook.com*, and is reflected in the data reported here.

The distribution of dynamic pages differs across websites. For *wikipedia.org*, we are not able to observe any calls to *eval* or run-time loaded code; therefore, no additional points-to pairs are found by blended analysis. For other sites, blended analysis adds between 1.4% (*blogger.com*) and 9.8% (*yahoo.com*) points-to pairs not found by pure static analysis.

Figure 5 presents the histogram of those pages that produce extra points-to pairs found by blended analysis. The two leftmost bars show that 68.7% of these pages add fewer than 4% of the points-to pairs to the entire program solution; however, 22.6% of the pages add more than 8% of the points-to pairs. The results suggest that dynamically 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 [22].

We have averaged these results over all the pages in each website. Though these averages do not imply that use of *eval* in JavaScript affects the points-to solution dramatically, there are some cases where the size of the string literal within the *eval* dominates the size of the program. Among the top 17 websites in *Alexa, sina.com.cn* contains a page with code size of 6KB in which there are 4KB Strings to be evaluated dynamically. For this page, the number of points-to pairs in the entire program solution increases by 60.1% by handling *eval*.

There is little difference in the ability to find extra points-to pairs between the original and optimized blended analyses; this is because the optimization criteria weights coverage of dynamic code heavily. The results in column 5 of Table 2 show that the two blended analyses agree in 4 out of 8 cases (shown in boldface), and their maximum average difference is 1% of the solution.

Pruning. Pruning was applied over all the 4695 traces we collected. On average over all traces, 23.9% of the basic blocks were pruned. Removing these basic blocks means the *Static Analyzer* has less code to analyze for each trace so that it can run faster. We found 354 variadic functions in total (Table 1). We observe 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*.

Timings. Table 3 presents the time performance of blended analysis on average over the pages on each website. We compare the analysis time of the pure static, original blended, and optimized blended analyses. The analysis time of blended analysis includes the time for collecting dynamic information, (optional) optimization, and the combined time of static analysis on each execution. Pure static analysis is 54.1% faster than the original blended analysis on average across all websites. (Recall that pure static analysis only analyzes statically accessible JavaScript code.) In a blended analysis, the static analysis phase is more costly than execution of the lightweight instrumentation. The main reason for the comparative slowness of the blended analysis versus the static is that some traces on a page may overlap in method coverage, causing those methods to be statically analyzed multiple times. The heuristic criterion in the optimized blended algorithm is designed, in part, to prevent this situation.

Summary. The timing results show that the optimized blended analysis is 25.7% faster than the original blended analysis, adding on average an overhead of 31.6% to static analysis, while accounting for *missing* points-to pairs from dynamically generated code and not significantly degrading accuracy on the statically accessible code.

Threats to validity The results of blended analysis heavily depend on the static analysis algorithm used. We have used the built-in Andersen-like points-to analysis algorithm in *WALA*. Therefore, the accuracy of our implemented framework is determined by limitations of the *WALA* analysis of JavaScript. We found that there were some parsing problems with some JavaScript code in the websites, and some structures of JavaScript were ignored.

Because we collected the execution traces of websites ourselves, there is a threat to validity of biasing the pages explored to find more dynamic features. Therefore, in the experiment, one author collected executions without knowledge of the website implementation.

6. RELATED WORK

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

Blended analysis of Java. Blended analysis of Java [6, 7] 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 aims to obtain a 'good enough' whole-program solution, by analyzing a set of executions. 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 and/or 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 [23]. 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 [22] show that the *eval* construct, which can generate code at runtime, is

Site	Static	Original blended			Optimized blended				
		dynamic	static	total	dynamic	optimization	static	total	overhead
GOGL	78.4	4.7	163.1	167.8	4.7	8.4	92.0	105.1	34.1%
FBOK	116.8	8.2	314.3	322.5	8.2	3.5	120.6	132.3	13.3%
YTUB	49.3	6.1	88.2	94.3	6.1	2.7	67.2	76.0	54.2%
YHOO	41.0	4.2	68.9	73.1	4.2	2.4	50.1	56.7	38.3%
BADU	28.4	2.3	38.1	40.4	2.3	1.3	32.2	35.8	26.0%
WIKI	20.3	2.5	24.2	26.7	2.5	1.2	22.2	25.9	27.6%
LIVE	32.8	3.8	51.9	55.7	3.8	1.0	31.3	36.1	10.1%
BLOG	10.2	3.0	15.4	18.4	3.0	0.6	11.6	15.2	49.0%

Table 3: Analysis time comparison (in minutes)

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 [21]. 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 JavaScript blended points-to analysis on website codes.

Static analysis. Various static analyses have been applied to JavaScript. Guarnieri, *et al.* [11] 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* could not be 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 [15] 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 [9]. 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.

Several type analyses have been proposed [16, 14, 1, 18]; however, JavaScript's dynamism makes it hard to achieve good precision. These approaches work on subsets of JavaScript (e.g., *JavaScript*$^{=}$ [18] 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.* [5] presented an information flow analysis for JavaScript. Guarnieri and Livshits [10] provided GULFSTREAM as a staged points-to analysis for streaming JavaScript applications. JavaScript blended analysis differs from their approaches in two ways. Instead of an incremental analysis, blended analysis collects dynamically generated/loaded code during profiling and it 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 [20]. 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 [8, 13] 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 general-purpose analysis framework on which we can build all kinds of client applications (e.g., for security and optimization).

7. 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 have defined the blended analysis algorithm and discussed the safety of its solution. We have implemented blended points-to analysis of JavaScript to evaluate our framework. Blended points-to analysis covers 86.6% of the pure static points-to solution on average and adds an average of 7.0% of the points-to pairs to the actual points-to solution, at a cost of 31.8% overhead over pure static analysis. Thus, the experimental results show that blended analysis is a practical 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 want to apply blended analysis to a points-to client problem such as finding security vulnerabilities or doing performance diagnosis.

8. REFERENCES

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

[2] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. Vex: vetting browser extensions for security vulnerabilities. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 22–22, Berkeley, CA, USA, 2010. USENIX Association.

[3] 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, Berkeley, CA, USA, 2009. USENIX Association.

[4] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th international conference on Static analysis*, SAS'03, pages 1–18, Berlin, Heidelberg, 2003. Springer-Verlag.

[5] 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, New York, NY, USA, 2009. ACM.

[6] 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, New York, NY, USA, 2007. ACM.

[7] 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, New York, NY, USA, 2008. ACM.

[8] 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, New York, NY, USA, 2009. ACM.

[9] 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, Berkeley, CA, USA, 2009. USENIX Association.

[10] 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, Berkeley, CA, USA, 2010. USENIX Association.

[11] 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, New York, NY, USA, 2011. ACM.

[12] 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, New York, NY, USA, 2009. ACM.

[13] 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, New York, NY, USA, 2011. ACM.

[14] 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, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] 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, New York, NY, USA, 2009. ACM.

[16] 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, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for java. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, APLAS'05, pages 139–160, Berlin, Heidelberg, 2005. Springer-Verlag.

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

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

[20] 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.

[21] 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, Berkeley, CA, USA, 2010. USENIX Association.

[22] 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, Berlin, Heidelberg, 2011. Springer-Verlag.

[23] 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, New York, NY, USA, 2010. ACM.