Determinacy in Static Analysis for jQuery

Esben Andreasen Anders Møller
Aarhus University
{esbena,amoeller}@cs.au.dk

Abstract

Static analysis for JavaScript can potentially help programmers find errors early during development. Although much progress has been made on analysis techniques, a major obstacle is the prevalence of libraries, in particular jQuery, which apply programming patterns that have detrimental consequences on the analysis precision and performance.

Previous work on dynamic determinacy analysis has demonstrated how information about program expressions that always resolve to a fixed value in some call context may lead to significant scalability improvements of static analysis for such code. We present a static dataflow analysis for JavaScript that infers and exploits determinacy information on-the-fly, to enable analysis of some of the most complex parts of jQuery. The analysis combines selective context and path sensitivity, constant propagation, and branch pruning, based on a systematic investigation of the main causes of analysis imprecision when using a more basic analysis.

The techniques are implemented in the TAJS analysis tool and evaluated on a collection of small programs that use jQuery. Our results show that the proposed analysis techniques boost both precision and performance, specifically for inferring type information and call graphs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Languages, Algorithms, Verification

Keywords JavaScript, program analysis

1. Introduction

JavaScript programmers need better tools to help detect errors early during development, transform code for refactoring or optimization, and understand existing code bases.

OOPSLA '14, October 19–21, 2014, Portland, OR, USA.

ACM 978-1-4503-2585-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2660193.2660214

```
1 jQuery.each("ajaxStart ajaxStop ... ajaxSend".split(" "),
2   function(i, o) {
3     jQuery.fn[o] = function(f) {
4       return this.on(o, f);
5   };
6 });
```

Figure 1. A small example of code (abbreviated with "`...`") from jQuery-1.8.0 that causes challenges for static analysis.

Static analysis may be a fruitful foundation for such tools. However, the dynamic language features of JavaScript are challenging to reason about with static analysis, and the prevalent use of libraries is a major obstacle for further progress.

Most JavaScript web applications today use libraries to provide convenient functionality on top of the basic browser API, and these libraries generally exploit the dynamic language features intensely. Moreover, the library code for many websites is an order of magnitude larger than the application code itself. A survey has shown that 58% of the top 10 million websites use the jQuery library, and it has a market share among JavaScript libraries of 93.4% [35]. This means that practical static analysis tools for JavaScript web applications must be able to cope with jQuery. Since jQuery evolves and new versions appear regularly, and thousands of plugins exist, writing analysis-specific models of the library, for example as detailed annotations of the library interface, is not a viable option. Instead, we need to improve the static analysis techniques to become able to reason precisely and efficiently about the programming patterns that are used in the library code.

As an example, consider what is required for a static analysis to reason about the small snippet of jQuery library code shown in Figure 1. This code converts a string into an array `["ajaxStart", "ajaxStop", ..., "ajaxSend"]` and then iterates over this array using the `jQuery.each` library function, assigning a function to a property of the object `jQuery.fn`, which jQuery applications use as prototype object for HTML node sets. In the first iteration, for example, `jQuery.fn.ajaxStart` is set to a function that passes a callback `f` to `this.on("ajaxStart",f)`. If the analysis does not have precise information about the value of `o` in the assignment `jQuery.fn[o]` or at the inner expres-

```
1  // Multifunctional method to get and set values of a collection
2  // The value/s can optionally be executed if it's a function
3  access: function(elems, fn, key, value, chainable, emptyGet, pass) {
4    var exec, bulk = key == null, i = 0, length = elems.length;
5    // Sets many values
6    if (key && typeof key === "object") {
7      for (i in key) {
8        jQuery.access(elems, fn, i, key[i], 1, emptyGet, value);
9      }
10     chainable = 1;
11   // Sets one value
12   } else if (value !== undefined) {
13     // Optionally, function values get executed if exec is true
14     exec = pass === undefined && jQuery.isFunction(value);
15     if (bulk) {
16       // Bulk operations only iterate when executing function values
17       if (exec) {
18         exec = fn;
19         fn = function(elem, key, value) {
20           return exec.call(jQuery(elem), value);
21         };
22       // Otherwise they run against the entire set
23       } else {
24         fn.call(elems, value);
25         fn = null;
26       }
27     }
28     if (fn) {
29       for (; i < length; i++) {
30         fn(elems[i], key,
31            exec ? value.call(elems[i], i, fn(elems[i], key)) : value,
32            pass);
33       }
34     }
35     chainable = 1;
36   }
37   return chainable ?
38     elems :
39       // Gets
40     bulk ?
41       fn.call(elems) :
42       length ? fn(elems[0], key) : emptyGet;
43 }
```

Figure 2. The jQuery `access` function, showing heavy use of overloading.

sion `this.on(o, f)`, then subsequent calls to the functions on `jQuery.fn` will be mixed together by the analysis. However, that information requires detailed knowledge by the analysis of the meaning of not only the `split` and `jQuery.each` functions, but also context sensitive reasoning of the nested closures. Specifically, notice that `o` in the innermost closure is bound in the outer closure, so the analysis must track the connections between the closure objects. In addition, the analysis must have precise knowledge of the values of `jQuery.each` and `split` at the function calls, which is not trivial to obtain due to the dynamic resolution of variables and object properties in JavaScript. In principle, both functions could be redefined elsewhere in the program. The `split` function is part of the JavaScript standard library [6]. The `jQuery.each` function, on the other hand, is itself defined in jQuery, using several `for` and `for-in` loops and the native functions `call` and `apply`. Those functions provide functionality to call any given function. Imprecise treatment of their arguments during the analysis of `each` will also have severe consequences on the analysis precision for the code in Figure 1.

JavaScript libraries often use reflection to implement overloading. As an example, the jQuery `attr` function for accessing HTML attributes works as a getter if given one string argument (the attribute name) and as a setter if given one object argument (with a set of properties corresponding to the attributes to set) or two arguments where the first is a string (the attribute name) and the second is a string (the new attribute value) or a function (that computes the new attribute value based on the old value). Much of that functionality is shared by other jQuery functions, so it is placed in a general, highly overloaded function, named `access`, shown in Figure 2. Lines 6–9 handle the case of multiple setters using recursive calls, single setters are handled in lines 12–34, and the return value for ordinary getters is computed in line 42. The overloading is implemented using reflection in the branches, for example, line 12 checks whether the `value` parameter is provided, in which case it is a setter operation. The actual getting and setting of attributes is done via the `fn` function in lines 20, 24, 31, 41 and 42. Analyzing functions such as `access` evidently requires precise techniques to avoid mixing together the different modes of operation and the dataflow from different call sites.

Static analysis must inevitably approximate the dataflow, but for JavaScript even small losses of precision are amplified by the poor encapsulation mechanisms of the language. (JavaScript has no module system and no notion of private fields, although these features to some extent can be mimicked with closures.) This often occurs with dynamic property access, written x`[`y`]` (e.g. line 3 in Figure 1) where x is an expression that evaluates to an object and y evaluates to a string, possibly via coercion, that is interpreted as an object property name. If x evaluates to the global object, which plays a central role in JavaScript programs, and the value of y is unknown due to approximations, then a read operation z=x`[`y`]` may conservatively result in numerous built-in values—including the `eval` function and the `Function` function that make it possible to execute dynamically generated code, the `document` object that gives access to the entire HTML DOM, and other critical objects, such as `Array` and `Math`. If the program being analyzed subsequently uses z in function calls or property access operations then the analysis may conservatively infer that, for example, `eval` is being invoked or the core methods on arrays are being modified. Most of the resulting dataflow is likely spurious in such cases. For an assignment x`[`y`]`=f where the value of y is unknown, static analysis will infer that the assignment may overwrite any method of x, including `toString`, which is used in type coercions. For a call x`[`y`](...)` where x is a function object and the value of y is unknown, y could in principle be the string `"apply"`, in which case the function x is invoked instead of some function stored in one of x's properties. Obviously, such imprecision quickly renders the analysis results useless.

Static analysis tools often require more time and space when precision degrades. Conversely, improving analysis precision can cause significant improvements of time and

space requirements in practice, even though the theoretical worst-case complexity may be higher [21, 30, 33]. While working with specific analysis tools for JavaScript, we often observe that some programs can be analyzed with high precision using modest resources, whereas other programs cannot be analyzed at all due to cascading spurious dataflow, even with many GB of space and hours of running time. The use of challenging dynamic language features is more decisive for analysis success than the program size.

Our goal is to enable practical, sound static analysis of JavaScript programs that use the jQuery library. As a measure of analysis precision we consider (1) *type analysis*, which infers the possible types of all variables and expressions, (2) *call graph construction*, which infers the possible callees at every function call site, and (3) *dead code detection*, which may be useful for optimization of applications that do not use the entire library. As we focus on the jQuery library itself, in this work we take the first steps and consider only small client programs, not full-scale applications of the library. For the reasons discussed above, time and space efficiency is not our primary concern at this point, and we simply consider analysis executions that require less than one minute as tractable.

Although jQuery has been studied in previous work related to dataflow analysis [22, 29, 33], no existing static analysis has been shown capable of reasoning precisely about the programming patterns found in libraries such as jQuery. Among the most promising ideas is the observation by Schäfer et al. that determinacy information can be exploited in static analysis, most importantly to handle dynamic property access operations and overloaded functions [29]. An expression is said to be *determinate* if it always evaluates to the same value when executed at runtime. In practice, the most useful determinacy facts only hold in a given context, so all determinacy facts in the approach by Schäfer et al. are qualified with a complete call stack and with values of loop iteration variables. As an example, a dynamic analysis of the code from Figure 1 may tell us that the variable `o` always has the value `ajaxStart` in line 4 for the first iteration of the loop inside the `jQuery.each` function, and similarly for the other iterations. Such information may then be used by a static analysis to enable context sensitivity and loop unrolling at the critical places in the JavaScript code. We elaborate on the connection between our approach and dynamic determinacy analysis in Section 2.

Our main contributions are as follows:

- We show how to integrate determinacy into a static analysis, which avoids the drawbacks of dynamic determinacy analysis. The design of our static analysis is based on a systematic investigation of imprecision in an existing dataflow analysis tool, TAJS [15–18], when applied to jQuery. By extending TAJS with selective context and path sensitivity, we obtain a fully automatic static analysis that simultaneously infers and exploits determinacy information. As part of this, we introduce a flow-sensitive variant of the correlation tracking technique by Sridharan et al. [33]. The individual analysis techniques we apply are variations of well known ideas, but to our knowledge they have not before been applied in concert to analyze JavaScript programs.
- We experimentally measure the effect of the analysis extensions on analysis precision and performance using a collection of small jQuery programs and different versions of the library. The experiments demonstrate that the improved analysis is capable of performing sound and precise analysis of 86 of 154 benchmark programs, showing that the presented techniques make it possible to obtain detailed type information and call graph information for substantial parts of jQuery. Most importantly, the effects of the individual analysis techniques are investigated. We observe that the combination of selective context and path sensitivity, constant propagation, and branch pruning is critical for successful analysis.

Although this work does not solve all the problems in static analysis of programs that use jQuery, we have now reached the important milestone of handling at least small applications that involve considerable parts of the library. Programs that could not be analyzed before with available resources can now be analyzed with high precision within seconds. These results not only provide new insight into challenges and possible solutions to static analysis for dynamic languages; they may also enable new practical tools for supporting JavaScript programmers become more productive.

We first, in Sections 2 and 3, describe the main related work on static analysis for JavaScript and jQuery and briefly introduce the existing TAJS analysis. Section 4 describes our methodology that has led us to the proposed analysis extensions, which are then explained in Sections 5–8. The experimental evaluation is described in Section 9.

2. Related Work

Static analysis for JavaScript has been studied for a decade [2, 34], initially considering only subsets of the language and ignoring practical issues, such as the browser environment and the programming patterns used in libraries. It has later become clear that handling real-world libraries, in particular jQuery, is "a formidable challenge to static analysis" [33].

Except for the TAJS analysis tool, described separately in Section 3, most analysis techniques separate pointer analysis from dataflow analysis. WALA from IBM Research [29, 33] is based on Andersen-style pointer analysis. Although the correlation tracking technique by Sridharan et al. [33] shows improvements, their experiments show that WALA is not able to build a call graph for jQuery without manually modifying the core library function `extend` (which we return to in Section 6) since "any sort of traditional flow-insensitive

analysis of this function gets hopelessly confused about what is being copied where". In addition, they report that WALA cannot analyze jQuery unless ignoring the built-in functions `call` and `apply`, which play an important role in jQuery. Those experimental results are for a JavaScript application that does nothing but load jQuery. By incorporating determinacy information, Schäfer et al. [29] take a step further, but still conclude that jQuery 1.3 and later versions are "not yet analyzable" with WALA, and even the older versions of jQuery require unsound modeling of the HTML DOM. Their technique works in two phases: First, they infer determinacy facts using a dynamic analysis. Each determinacy fact consists of (1) an expression e in the program, (2) a concrete value v, and (3) a call stack c including values of iteration variables. The interpretation of such a fact is that e will always have the value v when executed in a context that matches c. This is exploited in the second phase, which is a context sensitive static analysis. For example, if analyzing a read operation `x[y]` in a call context c and a determinacy fact provides the string value `"p"` for `y` in a context that matches c, then the analysis can treat the operation `x[y]` as `x.p` and hence know precisely which object property is being accessed.

The concept of determinacy plays a central role in our work, as discussed in Section 1. As mentioned, Schäfer et al. compute determinacy facts using a dynamic analysis before performing the static analysis of interest. In contrast, our analysis infers determinacy facts on-the-fly, during the static analysis. This has several advantages:

1. Since the determinacy facts being inferred by the dynamic approach are qualified with the entire call stack and values of loop iteration variables, each such fact only provides information for a very specific context. This means that it generally requires an extraordinarily thorough dynamic execution to cover all the contexts that may be relevant for the subsequent static analysis. The jQuery case study [29] eschews this problem by considering only the simplest possible jQuery application that loads jQuery and does nothing else, which means that a single execution of that application covers practically all relevant contexts. Our static analysis avoids explicitly computing such qualified determinacy facts and automatically covers all possible contexts.

2. Dynamic determinacy analysis requires a so-called *counterfactual execution* to account for branches that are not reached through the ordinary execution. This is not only nontrivial to implement; it also leads to imprecise results when indeterminate calls or native functions with side effects are encountered. For instance, a method call `x.m()` causes the entire heap to be marked as indeterminate by the dynamic analysis if `x` is not itself determinate. In contrast, a static analysis can simply model the possible callees according to the current abstract state, without losing all determinacy information.

Other related work includes the Actarus analysis by Guarnieri et al. [11], which performs taint analysis for JavaScript. It relies on WALA's pointer analysis and thereby inherits its limitations regarding jQuery. The Gatekeeper analysis by Guarnieri and Livshits [10] is also based on a variant of Andersen-style pointer analysis, using plain allocation site abstraction and a coarse modeling of dynamic property accesses and `for-in` loops, and it has not been shown capable of handling the complexity of the jQuery code. The related approach by Jang and Choe [14] has similar limitations.

Pointer analysis for JavaScript has been used as a foundation for refactoring. Again, using traditional pointer analysis techniques has shown to be insufficient for JavaScript libraries [7, 8].

The information flow analysis by Chugh et al. [5] uses a context-insensitive constraint-based technique. Dynamic property accesses are treated unsoundly, and the analysis lacks support for `call` and `apply`, so it is unlikely that such analysis is capable of reasoning precisely about dataflow in jQuery.

The control flow analysis by Guha et al. [12] is based on the uniform k-CFA algorithm. Some of their experiments involve the Prototype library, which causes similar difficulties for analysis as jQuery, and they had to manually modify 200 lines in the library to make it amenable to their analysis.

The event handler race analysis by Zheng et al. [39] achieves scalability by avoiding the jQuery code entirely and instead requiring manually written inference rules to model its functionality.

The analyses by Hackett and Guo [13] and Logozzo and Venter [23] are able to infer type information for optimization, but in connection with runtime execution, not using offline static analysis. The blended analysis by Wei and Ryder similarly handles the dynamic features of JavaScript by requiring a dynamic analysis to provide information for the static analysis [37].

Other techniques are able to analyze jQuery applications statically by sacrificing soundness, for example, ignoring dynamic property accesses altoghether [9], or ignoring all library code [24]. Another approach to reason about jQuery code is to introduce a static type system, as in TypeScript [38] or the work by Lerner et al. [22], however, we aim for a technique that does not require tedious and error prone maintenance of complex type annotations.

In summary, no existing static analysis can reason precisely about the programming patterns found in libraries such as jQuery. To this point, the state of the art is WALA, which is capable of analyzing a JavaScript program that does nothing but load jQuery, and only for outdated versions of jQuery with manual modifications and unsound treatment of the DOM and some core JavaScript functions.

3. Background: The TAJS Static Analysis

We assume that the reader is familiar with the basic features of the JavaScript programming language, and we here briefly describe the TAJS analysis tool [15–18] that our work is based on.[1]

TAJS is a whole-program dataflow analyzer for JavaScript, including the ECMAScript standard library [6] and large parts of the W3C browser API and HTML DOM functionality. JavaScript programs are represented in TAJS as flow graphs, with nodes representing primitive instructions and edges representing intraprocedural control flow. Interprocedural control flow is discovered during the analysis. A representative list of the primitive instructions that may appear at flow graph nodes are shown in Figure 3. (To simplify the presentation we here omit instructions related to deletion of variables and properties, `with` and `for-in` blocks, and special forms of call and construct operations.) Nested JavaScript expressions are linearized using a notion of temporary registers to hold intermediate values.

The dataflow analysis, which is derived from the monotone framework [19], is flow sensitive and partly context sensitive, using allocation site abstraction for objects and constant propagation for primitive values. The basic abstract domains used by the analysis, which closely mimic the ECMAScript specification, are shown in Figure 4. The flow graph for the program being analyzed defines the set N of flow graph nodes and the set R of temporary registers. The set C is used for context sensitive analysis, as explained in the following sections, L is the set of abstract addresses of objects. The set P contains all object property names, in other words, all strings plus some pseudo-properties: the internal ECMAScript properties [[Prototype]] and [[Value]], and *default_index*, and *default_other* that provide default values for, respectively, array index properties and other properties.

The main domain AnalysisLattice provides a call graph and an abstract state for each pair $(c, n) \in C \times N$ of a context and a flow graph node. A call graph $g \in$ CallGraph is a set of call edges, each defined by a caller context, the location of a call or construct node, a callee context, and a function entry node. An abstract state $s \in$ State provides an abstract object for each address, an abstract value for each register, and an execution context. An execution context in JavaScript contains the current scope chain (which is used for resolving variables) and references to a designated variable object (which is used for variable declarations) and a 'this' object (which defines the current value of `this`). Execution contexts are modeled by the lattice ExecutionContext. An abstract object $o \in$ Obj assigns an abstract value, absence information, and attribute information to every property name. (For property names where the abstract value is $\bot$, the information for *default_index* or *default_other* is used instead.)

[1] Our starting point is TAJS v0.9-2, which contains minor changes compared to the existing research papers on TAJS.

declare-variable$[x]$: declares a program variable named x with initial value `undefined`

declare-function$[f, r]$: creates a closure for the function f to be stored in register r

read-variable$[x, r]$: reads the value of a program variable named x into register r

write-variable$[r, x]$: writes the value of register r into a program variable named x

constant$[c, r]$: assigns a constant value c to register r

read-property$[r_1, r_2, r_3]$: performs an object property read where r_1 holds the base object, r_2 holds the property name, and r_3 gets the resulting value

write-property$[r_1, r_2, r_3]$: performs an object property write where r_1 holds the base object, r_2 holds the property name, and r_3 holds the value to be written

if$[r]$: represents conditional flow for `if`, `for`, and `while` statements

call$[v, r_0, \ldots, r_n, u]$ and construct$[v, r_0, \ldots, r_n, u]$: represent calls and `new` expressions, where register v holds the function value, $r_0, \ldots, r_n$ hold the values of `this` and the parameters, and u gets the return value

return$[r]$ and return-exc$[r]$: the unique exit (normal/exceptional) of a function body

throw$[r]$ and catch$[x]$: represent `throw` statements and entries of `catch` blocks

<*op*>$[r_1, r_2]$ and <*op*>$[r_1, r_2, r_3]$: represent unary and binary operators, where the result is stored in r_2 or r_3, respectively

Figure 3. The main instructions in TAJS flow graphs.

The sub-lattices Absent and Attr indicate whether the property may be absent, and, if present, what attributes (ReadOnly, DontDelete, DontEnum) it may have, and Mod tracks whether the property has been modified within the current function (see [15] for details). Each abstract object also carries a scope chain, represented as a finite list of sets of object addresses. Values are modeled by the product lattice Value with a component for each kind of value: `undefined`, `null`, booleans, numbers, strings, and references to objects. For each kind of primitive value, the corresponding lattice is chosen as the constant propagation lattice [36], with $\top$ representing any possible value and $\bot$ representing the situation where the value cannot have that type. The Num and String lattices, modeling numbers and strings, have additional lattice elements for representing unknown array index values. The String lattice furthermore has special lattice elements that group strings with a common prefix [18]. Note that TAJS models program variables as properties of activation objects, directly corresponding to the ECMAScript specification [6]. The transfer functions for the flow graph nodes model the effects of the instructions, including type coercions.

N : flow graph nodes

R : registers

C : contexts

L : object addresses

P : property names

$$\mathsf{AnalysisLattice} = (C \times N \to \mathsf{State}) \times \mathsf{CallGraph}$$

$$\mathsf{CallGraph} = \mathcal{P}(C \times N \times C \times N)$$

$$\mathsf{State} = (L \to \mathsf{Obj}) \times \mathsf{Registers} \times \mathsf{ExecutionContext}$$

$$\mathsf{Registers} = R \to \mathsf{Value}$$

$$\mathsf{ExecutionContext} = \mathsf{ScopeChain} \times \mathcal{P}(L) \times \mathcal{P}(L)$$

$$\mathsf{ScopeChain} = \mathcal{P}(L)^*$$

$$\mathsf{Obj} = (P \to \mathsf{Value} \times \mathsf{Absent} \times \mathsf{Attr} \times \mathsf{Mod}) \times \mathsf{ScopeChain}$$

$$\mathsf{Value} = \mathsf{Undef} \times \mathsf{Null} \times \mathsf{Bool} \times \mathsf{Num} \times \mathsf{String} \times \mathcal{P}(L)$$

Figure 4. Basic abstract domains used by TAJS.

In its most basic mode, TAJS uses allocation site abstraction where objects are distinguished only by their allocation site, that is, $L = N$. The default behavior, however, adds recency abstraction [3] to distinguish the most recently allocated object from older ones originating from the same allocation site, thereby enabling strong updates for many write operations. Since a single primitive instruction may create multiple new objects, for example at call instructions, TAJS further distinguishes between objects of different kinds, for example, activation objects and arguments objects.

TAJS employs a basic form of path sensitivity by pruning certain infeasible dataflow at branches [1]. For example, at a conditional statement `if (x)` S_1 `else` S_2, the analysis may assume that `x` is not `false`, `undefined`, or any other "falsy" value at the entry of S_1 and conversely at S_2. In situations where pruning infeasible values leads to $\bot_{\mathsf{Value}}$ (for example, if `x` is constant) the entire abstract state can be reduced to $\bot_{\mathsf{State}}$, thereby concluding that the corresponding branch is unreachable in the given call context.

The default setting for context sensitivity is based on the idea of object sensitivity [26], using $C = \mathcal{P}(L)$, where call contexts are distinguished by the abstract value of `this` (but only for functions that contain `this`).

Other analysis techniques used by TAJS include lazy propagation [16], abstract garbage collection [25], modified flags [15], modeling of the HTML DOM and browser API [17], and on-the-fly elimination of `eval` calls [18], but those techniques are not essential to understand the extensions we consider in this paper.

4. Integrating Determinacy in Static Analysis

The concept of determinacy fits naturally into static analysis, as known from context sensitive constant propagation for primitive values and single-target resolution for objects in many existing analyzers. A determinacy fact in static analysis is then simply a dataflow fact that an abstract value in the abstract state at a given flow graph node and context is a single, concrete value, such as, a string constant or a function object. Thus, constant propagation is in itself a simple form of determinacy analysis, which TAJS already exploits at dynamic property access (by treating x`[`y`]` as x`.`p if y is known to be a constant string p) and at conditionals (by pruning infeasible flow, as mentioned in the previous section).

This raises two questions: (1) It is possible to extend TAJS to infer more determinacy information? (2) Can we exploit that additional information to improve analysis of jQuery? Our starting point is an analysis that suffers from fatal imprecision and extreme memory and time consumption when given a trivial JavaScript application that loads jQuery and does nothing else. We now present some simple techniques that have helped us reveal the main causes of imprecision or redundant work in the analysis, to suggest opportunities for inferring and exploiting more determinacy information.

First, we interrupt the dataflow fixpoint solver after a number of iterations, e.g. 10 000 node transfer function executions. Since the fixpoint has not been reached, the current abstract states are generally not a sound approximation of the possible program behavior, however, they can still be useful for identifying likely spurious dataflow. We define a heuristic measure of *suspiciousness* on the abstract values and the call graph:

- An abstract value is suspicious if it has many different types, where we categorize types as numbers, strings, booleans, functions, arrays, native objects, DOM objects, and other objects.
- A call graph is suspicious at a given call site and call context if that pair has a large number of callees.

We have found that a high degree of suspiciousness often indicates a high amount of spurious dataflow, so manually inspecting the most suspicious cases often provides valuable insights about the causes of imprecision and potential solutions. Using a delta debugging tool, such as JS Delta[2], with a suspiciousness threshold as predicate helps finding smaller programs that exhibit the bottlenecks of interest. An additional effective technique to reduce the complexity of the task is to apply a simple form of program slicing by removing the functions and conditional branches in the jQuery code that are not used by the given application at runtime.

Based on such an investigation of imprecision when attempting to analyze programs that use jQuery, we propose

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

three analysis improvements related to context and path sensitivity: *parameter sensitivity* as a variant of object sensitivity, *loop specialization* to improve precision of `for` loops and `for-in` loops, and *context sensitive heap abstraction* for selected allocation sites. Each improvement can be expressed and implemented as a manageable extension of the analysis framework described in Section 3. Specifically, the abstract domains only require modifications of C and L.

5. Parameter Sensitivity

As indicated by the examples in Figures 1 and 2, it is critical that certain library functions are analyzed context sensitively. Object sensitivity is not sufficient, since `this` is not used by many of the functions. An alternative, such as 1-CFA [32], which includes the call site in the context, is also insufficient, since many of the library functions, including `each` and `access` from our examples, involve multiple layers of nested calls. Although the more general technique k-CFA (or, the call string technique [31]) may help, our study of imprecision, as discussed in Section 4, suggests that we instead focus on the abstract values of selected function parameters. For example, what matters for a call to the closure in line 2 in Figure 1 is not the location of the call site but the values of its two parameters. For this reason, we suggest a notion of *parameter sensitivity*, which can be seen as a variant of object sensitivity that is easily incorporated into the existing TAJS analysis framework. The basic concept of parameter sensitivity is not new – similar techniques have been used in pointer analysis [21] and type inference [27] – but it has not been investigated before how it works for JavaScript libraries and when and how it should be applied.

The idea in selective parameter sensitivity is that certain functions should be analyzed context sensitively depending on the abstract values of their parameters at the call sites. If we let A denote the set of all function parameters (identified by their position in the list of actual parameters) we now extend the notion of contexts from $C = \mathcal{P}(L)$ to:

$$C = \mathcal{P}(L) \times (A \rightharpoonup \mathsf{Value})$$

The interpretation of a context $(t, a) \in C$ is that t represents the value of `this`, as in Section 3, the extra component a specifies an abstract value for selected parameters in A for the current function. The analysis transfer functions require modifications only for call and construct: when creating a context $(t, a) \in C$, the t component is created as before, and the a component is found by reading the abstract values of the selected actual parameters in the current abstract state. With this infrastructure in place, the question that remains is how to decide which parameters to select when constructing the new contexts. Selecting all parameters will likely be too costly and result in an explosion in the number of contexts; selecting none is equivalent to omitting the analysis extension. Based on the approach explained in Section 4 we propose the following simple heuristic that performs the selection on-the-fly, during the static analysis:

> When a transfer function creates a new context (t, a) at some call site, we select those parameters for inclusion in a whose abstract value is a concrete string (i.e., a known string in the String lattice) or a single object address (i.e., exactly one object allocation site).

According to the structure of the domain AnalysisLattice, this extension allows us to have multiple abstract states at a program point, irrespective of `this`. As an example, if the `access` function from Figure 2 is called twice, at each place with a function literal as the second parameter (named `fn`), then the body of the `access` function will effectively be analyzed twice, which prevents the two function literals from being mixed together. In this way, the analysis knows precisely what `fn` refers to in each context. This increases precision at all calls to `fn` inside `access`, and it also works smoothly in line 8 where `fn` is passed as a parameter in a recursive call.

Regarding the example from Figure 1, the use of parameter sensitivity will allow the analysis to distinguish between the different values of `o` in line 3. However, more precision improvements are needed to handle the entire example satisfactorily, which we return to in Section 7.

Notice how TAJS's constant propagation and context sensitive interprocedural analysis fit well together. All the functions that appear in Figure 1 may be invoked with many different parameter values, but our selective use of parameter sensitivity ensures that the central parameters often have *constant values in each context*. In other words, the static analysis infers determinacy information that is qualified by abstract contexts, not by concrete call stacks as in dynamic determinacy analysis. Other static analysis algorithms perform context sensitive constant propagation [28]. In our setting, it is important that the constant propagation includes folding at operators, for example == and === in Figure 2, and at native ECMAScript functions, for example `split` in Figure 1, which we discuss in Section 8.

As the extension with parameter sensitivity makes the analysis lattice infinite-height (since the Value lattice that we now use in C has infinite width), some form of widening is needed to ensure termination. We simply select a threshold, for example 100, specifying the maximum number of abstract states at any program point. If this limit is exceeded, parameter sensitivity is disabled for the functions in question. However, this is mostly a theoretical concern: if the number of abstract states grows large, the analysis will likely be imprecise and not terminate anyway within a reasonable time budget. The worst-case complexity of the analysis increases dramatically by this extension. The interesting question is whether this complexity outweighs the increased precision, which we study experimentally in Section 9.

```
1  each: function(obj, callback, args) {
2    var name, i = 0, length = obj.length,
3      isObj = length === undefined || jQuery.isFunction(obj);
4    if (args) {
5      if (isObj) {
6        for (name in obj) {
7          if (callback.apply(obj[name], args) === false) {
8            break;
9        } }
10     } else {
11       for (; i < length;) {
12         if (callback.apply(obj[i++], args) === false) {
13           break;
14         }
15       }
16     }
17   // A special, fast, case for the most common use of each
18   } else {
19     if (isObj) {
20       for (name in obj) {
21         if (callback.call(obj[name], name, obj[name]) === false) {
22           break;
23         }
24       }
25     } else {
26       for (; i < length;) {
27         if (callback.call(obj[i], i, obj[i++]) === false) {
28           break;
29         }
30       }
31     }
32   }
33   return obj;
34 }
```

Figure 5. The jQuery `each` function.

6. Loop Specialization

Some central library functions, including the `each` function (called in line 1 in Figure 1 and shown in Figure 5) and also the `access` function (Figure 2) iterate through arrays, and we sometimes observe a critical loss of precision, in the form of an unknown dynamic property read, if the analysis does not distinguish between the individual iterations. For example, in the `for` loop starting in line 26 in Figure 5 it is important that the possible values of `i` are not mixed together. The callback is invoked with `obj[i]` as the parameter, so imprecision of `i` will propagate to, for example, `f` in line 4 in Figure 1. This makes the analysis unable to distinguish between the different functions in `jQuery.fn`. One consequence of this is that subsequent calls to `jQuery.fn.each` will produce dataflow to the function on line 3 in Figure 1, which will register the callback parameter as an event handler, ultimately resulting in a proliferation of spurious dataflow.

6.1 Specialization of `for` Loops

Our solution is to perform a simple kind of loop unrolling during the analysis. We do this by, again, extending the notion of contexts, adding yet another component to C:

$$C = \cdots \times (B \rightharpoonup \mathsf{Value})$$

Here, B is the set of local variable names occurring in the program, and "$\cdots$" corresponds to the former definition of C in Section 5. The interpretation of a context $(\ldots, b) \in C$ is that b describes the values of selected local variables. Notice that the notion of "context" now not only characterizes information from call sites, but also intraprocedural information. Whenever the analysis fixpoint solver propagates abstract states intraprocedurally, the contexts now change when the abstract values of the variables in b change. This is, fortunately, easy to accommodate in the TAJS implementation. Similar to our approach to parameter sensitivity, the question is which local variables to select when constructing b. We again propose a simple heuristic that decides on-the-fly:

> When a context $(\ldots, b)$ is created, we select those local variables for inclusion in b whose abstract value is a concrete integer (as represented in the Num lattice) in the current abstract state. However, we include only variables that appear syntactically in the condition and in the body of a non-nested `for` loop in the current function, and are involved in a dynamic property read operation (i.e. as a sub-expression of e_2 in e_1`[`e_2`]`).

Applying this technique to the `each` function in Figure 5, the analysis will select the local variable `i` for specialization and effectively unroll the `for` loops, creating a new context for every loop iteration. This can be viewed as a form of path sensitivity [4]. Observe how this analysis extension improves the inference of determinacy information, especially when combined with the parameter sensitivity technique from Section 5. The analysis exploits this information, for example to precisely model which parameter values are passed to the callback in line 27.

Some form of widening is obviously needed to ensure termination, as in Section 5. As a pragmatic solution, we choose to restrict this context specialization to a small range of numeric values, 0 to 50. Similar to the discussion about the increased theoretical worst-case complexity of adding parameter sensitivity, our experiments in Section 9 show whether the extra complexity caused by this additional extension pays off.

6.2 Specialization of `for-in` Loops

We now consider the `for-in` loops, which can often also benefit from context specialization as they play a central role in many library functions, including `access` (Figure 2), `each` (Figure 5), and `extend` (Figure 6). The `extend` function plays a central role in jQuery for populating its core objects, but it is also often used in application code to copy properties from one object to another. The first part (lines 2–23) inspect the `arguments` object to determine which of the many possible ways it is being used. In particular, this sets `target` to the object being copied to, and `i` becomes the index of the first parameter containing a source object. The outermost `for` loop (lines 24–49) then iterates through the source objects, and the innermost `for-in` loop (lines 28–49) iterates though their properties, copying them to the target object. The `deep` variable determines whether to use shallow or deep copying. Notice again how interprocedural constant propagation is exploited: calls to `extend` typically pass in

```
1  jQuery.extend = jQuery.fn.extend = function() {
2    var options, name, src, copy, copyIsArray, clone,
3      target = arguments[0] || {},
4      i = 1,
5      length = arguments.length,
6      deep = false;
7    // Handle a deep copy situation
8    if (typeof target === "boolean") {
9      deep = target;
10     target = arguments[1] || {};
11     // skip the boolean and the target
12     i = 2;
13   }
14   // Handle case when target is a string or something
15   // (possible in deep copy)
16   if (typeof target !== "object" && !jQuery.isFunction(target)) {
17     target = {};
18   }
19   // extend jQuery itself if only one argument is passed
20   if (length === i) {
21     target = this;
22     --i;
23   }
24   for (; i < length; i++) {
25     // Only deal with non-null/undefined values
26     if ((options = arguments[i]) != null) {
27       // Extend the base object
28       for (name in options) {
29         src = target[name];
30         copy = options[name];
31         // Prevent never-ending loop
32         if (target === copy) {
33           continue;
34         }
35         // Recurse if we're merging plain objects or arrays
36         if (deep && copy && (jQuery.isPlainObject(copy) ||
37                              (copyIsArray = jQuery.isArray(copy)))) {
38           if (copyIsArray) {
39             copyIsArray = false;
40             clone = src && jQuery.isArray(src) ? src : [];
41           } else {
42             clone = src && jQuery.isPlainObject(src) ? src : {};
43           }
44           // Never move original objects, clone them
45           target[name] = jQuery.extend(deep, clone, copy);
46         // Don't bring in undefined values
47         } else if (copy !== undefined) {
48           target[name] = copy;
49   } } } }
50   // Return the modified object
51   return target;
52 };
```

Figure 6. The jQuery `extend` function.

literal objects, which are then kept separate by the analysis due to the use of parameter sensitivity.

TAJS represents `for-in` loops using four special kinds of flow graph instructions as illustrated in Figure 7. Intuitively, begin-for-in$[r_1, r_2]$ creates an iterator r_2 for iterating through the properties of the object pointed to by r_1, has-next-property$[r_2, r_3]$ sets r_3 to true or false depending on whether r_2 has reached the end or not, next-property$[r_2, r_4]$ picks the next property from r_2 and stores its name in r_4, and end-for-in represents the end of the loop body. We use $r_1, r_2, r_3, r_4 \in R$ as temporary registers, as discussed in Section 3. The iteration order is implementation dependent according to the ECMAScript specification, so to remain conservative TAJS must model all possible orders. The ordinary behavior of TAJS is to let r_2 represent the least upper bound of the property names of r_1 (including `null` to represent end-of-list), set r_3 to any-boolean, and set r_4 equal to r_2, corresponding to nondeterministically picking a property

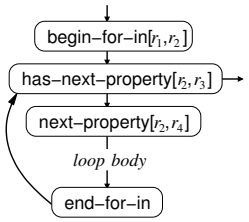

Figure 7. Representation of `for-in` loops in TAJS flow graphs.

name in each iteration. Although sound, this is evidently too imprecise for jQuery.

This motivates the addition of a variant of the context specialization mechanism that we introduced above for ordinary `for` loops, now considering selected registers instead of program variables using an additional component in C:

$$C = \cdots \times (R \rightharpoonup \mathsf{Value})$$

We now modify the transfer functions of the four special instructions as follows. First, the transfer function of begin-for-in will attempt to split the current context into a specialized context for each property name in the r_1 object, such that r_2 in the corresponding abstract state that is propagated to has-next-property is a fixed string representing exactly one of the property names. Again, an on-the-fly heuristic decides whether to perform this specialization or not:

> The transfer function of begin-for-in will split the current context if the set of property names of r_1 can be determined precisely in the current abstract state.

(Otherwise, it will use the ordinary behavior, as explained above, as a fallback.) For instance, if the current abstract state has $r_1 = \{\ell_7\}$ for some abstract address $\ell_7 \in L$ and the abstract object at ℓ_7 has the property names $\{\texttt{p1}, \texttt{p2}\}$ in that abstract state then the current context $(\ldots, d) \in C$ is split into two specialized versions, $(\ldots, d[r_2 \mapsto \texttt{p1}])$ and $(\ldots, d[r_2 \mapsto \texttt{p2}])$, that are propagated along with the correspondingly specialized abstract states to the has-next-property instruction. The analysis is then ready to analyze the loop body twice, once for each property name.

When the transfer function for has-next-property receives such a specialized abstract state where the value of r_2 is a fixed string p and not the end-of-list marker `null`, it will immediately pass the abstract state to the `next-property` instruction, which then assigns p to r_4. The only nontrivial case is the transfer function for end-for-in, which is responsible for merging the changes made to the abstract states by the loop body. Simply taking the least upper bound of the abstract states is a sound, but overly conservative solution, since it will effectively include the infeasible execution that skips the loop body entirely. Instead, we exploit the fact that TAJS already keeps track of which parts of the abstract states have been modified since the entry of the current function (cf. the Mod sub-lattice in Figure 4; see also [15, 16]).

By treating every has-next-property instruction as a pseudo-function entry, this directly gives us the heap locations that have been modified by the loop body. The merged abstract state created by end-for-in is then defined as the least upper bound of the specialized abstract states, except that we let modified parts in one state overwrite non-modified parts from another state. To soundly model the nondeterministic iteration order, we make sure the effects of one iteration are visible by the others by propagating the merged abstract state not only to the successor node of the loop but also back to the specialized contexts at the has-next-property node.

As a result, the analysis soundly captures all possible dataflow, without mixing together the property names. For example, the `extend` function from Figure 6 is used for bootstrapping the jQuery library with several calls of the form `jQuery.extend({`p_1`:`v_1`, ..., `p_n`:`v_n`})` that copy the given object properties to the `jQuery` object itself. Our analysis technique handles such patterns with high precision.

6.3 Connections to Other Analysis Techniques

The form of context specialization at `for-in` loops presented in Section 6.2 is reminiscent of the *correlation tracking* technique by Sridharan et al. [33] in WALA's flow insensitive pointer analysis. The primary motivating example considered by Sridharan et al. is a simple version of the `extend` function from jQuery. The key idea in correlation tracking is to identify correlated dynamic property accesses, such as `copy = options[name]` and `target[name] = copy` in Figure 6 (lines 30 and 48), extract the block of code containing the accesses into a new function, and then analyze that function context sensitively. The main differences between correlation tracking and our `for-in` specialization technique are that our approach works for flow sensitive analysis and without explicitly tracking correlated read/write pairs or extracting functions. Moreover, our approach does not require manual modifications of jQuery's `extend` function, unlike WALA.

We have seen in the preceding sections how the increased context sensitivity and loop specialization can boost constant propagation. This in turn gives more power to the basic form of path sensitivity that TAJS uses at branches as described in Section 3. Consider, for example, the branches in lines 8, 20, and 20 in Figure 6 for a call `jQuery.extend({`p_1`:`v_1`, ..., `p_n`:`v_n`})` with parameter sensitivity enabled. With that context, the analysis will infer that the three branch conditions will definitely evaluate to `false`, `false`, and `true`, respectively. In other words, the analysis automatically prunes branches that are infeasible in the specific contexts, thereby eliminating a considerable source of spurious dataflow. In this way, our combination of static analysis techniques can also be viewed as an alternative to the dynamic determinacy analysis discussed in Section 2: we effectively infer determinacy information during the static analysis rather than requiring a separate dynamic analysis.

The notion of type refinement by Kashyap et al. [20] is based on the same idea that TAJS uses at branch conditions to restrict dataflow, as mentioned in Section 3, although Kashyap et al. suggest more kinds of branch conditions than TAJS currently supports. Interestingly, we observe only negligible improvements of the analysis results for jQuery by using this technique, since it is often dominated by the use of context sensitive constant propagation and pruning of infeasible branches.

7. Context Sensitive Heap Abstraction

Our investigation of analysis imprecision, as described in Section 4, suggests that context sensitive heap abstraction [21] may also be necessary. Figure 1 again motivates the need: if analyzing a program that uses, for example, `jQuery.fn.ajaxStart` or `jQuery.fn.ajaxSend`, it is important that these functions are kept separate in the abstraction. Parameter sensitivity is not sufficient, because the functions are closures with free variables that are bound in enclosing activation objects.

We incorporate a variant of context sensitive heap abstraction by extending the definition of abstract addresses from $L = N$ to:

$$L = N \times (A \rightharpoonup \mathsf{Value}) \times (B \rightharpoonup \mathsf{Value}) \times (R \rightharpoonup \mathsf{Value})$$

(As mentioned in Section 3, to simplify the presentation we here ignore the other components of L that are used for recency abstraction and object kinds.) This means that objects are now not only distinguished by their allocation site, but also by a valuation of selected program variables and registers. Similar to our other analysis extensions, there is a wide range of possibilities for selecting when and how to apply this technique. Motivated by our study of analysis imprecision, we suggest using the extra components of L in four situations in different transfer functions:

1. When a declare-function instruction with source location n in some context $(t, a, b, d) \in C$ creates a new function object, its abstract address is selected as $(n, a, b, d) \in L$; that is, the context sensitivity valuations a, b, and d are taken directly from the current context. This causes, e.g., `jQuery.fn.ajaxStart` or `jQuery.fn.ajaxStop` to have distinct abstract values after the code in Figure 1 has been executed.
2. When a call or construct instruction creates a new activation object and a new arguments object, the abstract addresses of these objects obtain their context sensitivity valuation directly from the newly created callee context. This allows the analysis to distinguish the scope chains of the function objects for `jQuery.fn.ajaxStart` and `jQuery.fn.ajaxStop`. Thereby, when the functions `jQuery.fn.ajaxStart` or `jQuery.fn.ajaxStop` are later invoked, the value of `o` in line 4 in Figure 1 will be bound precisely to `"ajaxStart"` or `"ajaxStop"`, respectively, without mixing together the different values.

3. Wrapper objects created in type coercions are treated context sensitively in the same way as function objects, as explained above, except that we only use the parameter sensitivity component. As a motivating example, jQuery passes primitive string values via the `call` function in `each` (line 27 in Figure 5) into `this` in the callback whereby they are coerced into string wrapper objects.
4. For every object literal $\{p{:}v,\ \ldots\}$ that is in a `for-in` loop or syntactically contains a function parameter that is present in the parameter sensitivity component (Section 5) of the current context, the abstract address of the object will be context sensitive in the same way as wrapper objects. This is useful, for example, when jQuery creates its `innerHeight` method using nested calls to `each` and an object literal of the form `{padding: "inner" + name, ...}`.

For all other objects, the heap abstraction remains context insensitive. The experiments in Section 9 show the effect of this analysis extension on the precision and performance.

8. Modeling Standard Library Functions

The original version of TAJS models the JavaScript standard library functions with a focus on type information, which is the natural choice when developing a static type analysis [15]. As an example, `String.prototype.split` is modeled as creating an array of unknown strings. A more precise model is, however, needed when analyzing line 1 in Figure 1 and similar uses of `split` throughout jQuery. We observe a similar situation with regular expressions, which are used frequently in jQuery. For instance, jQuery contains the expression

```
dataTypeExpression.toLowerCase().split(core_rspace)
```

which TAJS reaches for an abstract state where the value of `dataTypeExpression` is the string `"json jsonp"` and `core_rspace` is the regular expression `/\s+/`. Those values originate from other functions, and our use of context sensitive interprocedural constant propagation is necessary to infer that the values are constants that for a specific context.

Our solution is, not surprisingly, to strengthen the models of the standard library functions to obtain full precision in cases where their input consists of constant primitive values in the abstract state at the call site. As a consequence, calls with determinate input will return determinate output. We achieve this by invoking an external JavaScript interpreter.[3]

Another critical source of imprecision is the use of object property names that consist of random sub-strings. Such property names are used by jQuery and other libraries to attach library data to non-library objects. The following two expressions from jQuery show how such property names are typically computed (the actual code varies slightly between the different versions of the library):

```
"jQuery" + (jQuery.fn.jquery + Math.random())
           .replace(/\D/g, "")

("sizcache" + Math.random()).replace(".", "")
```

The resulting strings are used as keys in dynamic property accesses. If the analysis models these strings conservatively as "any string" (i.e., the top element of the String lattice), then it will mix together the library data and all the other properties of the involved objects.

Our solution is to augment the Value lattice with a bounded number k of "random numbers or strings" and augment the State lattice with a counter from 0 to k. The transfer function for `Math.random` then picks the next random-value element and increments the counter if less than k. Note that the correctness of jQuery relies on the assumption that the randomly generated property names do not clash with any other properties. If that assumption holds for a million websites, we can use it too: in all comparisons, we can assume that these random values are different from all other strings and numbers, in particular, object property names. Similarly, we assume that operations, such as the + operator or the built-in `replace` function, produce random values when given random values as input. Since these random strings are always created in the library initialization phase, it suffices to set the bound k to a low number, e.g., 10.

9. Experimental Evaluation

Our main research question is: Do the analysis techniques suggested in Sections 5–8 improve the analysis efficacy? The extensions increase the theoretical worst-case complexity of the analysis, so it is not obvious whether they improve or degrade the results of the analysis. More specifically, for each of the extensions, does the theoretically increased precision cause more programs to be analyzable, or conversely, does the increased complexity manifest itself and diminish the increased precision? In cases where the analysis succeeds within a reasonable time bound, how precise is it? And conversely, in cases where it fails, is the cause related to our analysis extensions, or can those cases suggest directions for future work in improving the analysis further?

To evaluate the suggested analysis techniques and answer these questions we have implemented them in TAJS and collected four groups of benchmark programs: **(A)** The first group contains 12 programs that each load a different version of jQuery from 1.0.0 to 1.11.0 and does nothing else.[4] **(B)** The second group consists of 71 programs from a jQuery tutorial[5] that load jQuery 1.10.0[6] and perform a few simple operations using the library, thereby exercising the different categories of functionality in jQuery. **(C)** To be able to explore the different parts of jQuery in isolation we use the

[3] `http://www.mozilla.org/rhino/`

[4] jQuery version 2.x is different from 1.x in that compatibility support for older browsers has been removed, however, it uses ES5 getters and setters, which are not yet fully supported by TAJS.

[5] `http://www.jquery-tutorial.net/`

[6] jQuery 1.10.0 was the newest version when this work was initiated.

jQuery version	LOC	load-LOC	flow graph nodes
1.0.0	981	261	6 925
1.1.0	1 125	290	7 878
1.2.0	1 484	289	11 190
1.3.0	2 124	632	16 282
1.4.0	2 828	725	21 630
1.5.0	3 583	910	26 882
1.6.0	3 896	969	29 078
1.7.0	4 082	1 108	30 443
1.8.0	4 074	1 158	30 093
1.9.0	4 121	1 162	30 861
1.10.0	4 141	1 192	31 234
1.11.0	4 311	1 218	32 927

Table 1. The main versions of jQuery, showing the number of source lines that contain executable code, the number of lines containing code that is executed by loading the library in Chrome, and the number of primitive instructions in the TAJS flow graph.

71 programs from the jQuery tutorial again, but now using sliced versions of jQuery 1.10.0. That is, each program in group C uses its own sliced version of jQuery where all parts that are unreachable according to a dynamic execution in the Chrome browser have been removed. **(D)** Finally, to measure the impact of the analysis modifications on non-jQuery code, we include 69 programs from the Google Octane suite, the SunSpider suite, the 10K Event Apart Challenge, and Chrome Experiments that have been used in previous work on TAJS.

The jQuery benchmarks in groups A, B, and C are evidently far from full scale web applications, but as discussed in Section 2 just loading jQuery causes major complications for existing analysis tools. Some characteristics of the different versions of jQuery are shown in Table 1. In particular, the numbers in the table show that substantial parts of the library are involved in the loading process.

Our implementation and all benchmarks are available online.[7] All experiments are performed on a 2.9 GHz PC running a JVM with 2 GB RAM.

We first run TAJS on the benchmark programs from groups A, B, and C in 7 different analysis configurations. The first is the modified analysis with all tricks enabled:

- branch pruning (Section 3),
- parameter sensitivity (Section 5),
- loop specialization for ordinary loops (Section 6.1),
- loop specialization for `for-in` loops (Section 6.2),
- context sensitive heap abstraction (Section 7), and
- improved modeling of the standard library (Section 8).

[7] `http://brics.dk/TAJS/jquery.html`

	Programs	*Success*
A	12	11
B	71	27
C	71	48
total	154	86

Table 2. Analysis success for the three groups of jQuery benchmarks, with all analysis features enabled.

Each of the remaining six configurations disables a single of these analysis features. We classify an analysis execution as successful if it reaches a fixpoint within one minute.

The configuration with all features enabled results in successful analysis in 86 cases. The numbers for each of the three groups of benchmarks are shown in Table 2. This is a substantial improvement compared to WALA, which is unable to analyze beyond the first 3 versions of the 12 programs in group A, as mentioned in Section 2. Moreover, for all the remaining configurations we find that almost none of the 154 benchmark programs can be analyzed successfully. In other words, *all* the analysis features must be enabled to obtain successful analysis. This observation confirms that the individual analysis techniques support each other, as discussed in Section 6.3.

In the cases where the analysis succeeds, the analysis time is between 1 and 24 seconds, with an average of 6.5 seconds. Increasing the time-out by a factor 10 allows only three additional cases to succeed, which confirms that the analysis precision is more important than the time bound.

To measure the analysis precision we investigate how well TAJS performs type analysis, infers call graphs, and detects dead code for the 86 successful executions in the configuration where all features are enabled:

1. For every reachable read-variable and read-property instruction we count the number of possible types that the resulting value may have in each abstract state (using the same categories of types as in Section 4). According to the analysis, 99% of the values have a unique type. As the analysis is conservative this is a lower bound. Also, the analysis finds that the average number of types for each of the instructions is 1.006, where the real number must be at least 1.

2. We inspect the abstract values of all properties of the objects `jQuery` and `jQuery.fn` at the exit of the program. Those properties define the public interface of the library, so it is important that the analysis is capable of keeping the individual functions apart, which is challenging since they are created dynamically when the library is loaded. We find that 99% of the values that contain functions are resolved uniquely by the analysis.

3. We measure for each call and `construct` instruction in each reachable call context the number of possible callees. (Due to the use of higher-order functions, we choose to measure this per context instead of considering all possible callees for a given call.) As a result, 99% have a unique callee according to the analysis.

4. For each benchmark program in group A and B, we count the lines of dead code reported by TAJS relative to the actual dead code as observed by exhaustively executing the programs using the Chrome browser. On average, TAJS finds 98% of the actual dead code.

These results show that the analysis has high precision in the cases where it reaches a fixpoint before the time-out.

A concern about the benchmark programs that are *not* analyzed successfully is that the cause could in principle be explosions in the number of contexts or abstract addresses resulting from the increased worst-case complexity with the analysis extensions. To test this, we also measure the analysis precision using the same metrics as above, on abstract states obtained after the 1 minute time-out. The resulting abstract states are then under-approximations of the fixpoints, but they can still be useful for measuring precision (as discussed in Section 4). As a result we see in each of the failing cases that the cause is not an overwhelming number of contexts or abstract addresses, but imprecision in the individual abstract states. The following numbers are for the benchmarks in group C, which only contain reachable code and are thereby easier to compare:

- The average number of contexts and abstract addresses relative to the number of functions is 9.84 and 25.8, respectively, for the 48 successful cases. The corresponding numbers for the 23 unsuccessful cases at the time-out are not much higher: 14.7 and 31.5.
- The relative number of read-property instructions that yield abstract values with multiple possible types (measured as in Section 4) is 13 times higher for the unsuccessful cases than for the successful cases.

Although the proposed analysis techniques are obviously not sufficient to handle all aspects of jQuery, this indicates that the techniques are necessary constituents in jQuery static analysis tools.

To investigate this further, we also run the analysis on the 69 benchmarks in group D that do not use jQuery. Each of those benchmarks is analyzable with the original TAJS. Enabling all the new features except loop specialization for ordinary loops (Section 6.1) improves not only precision but, surprisingly, also the analysis time. In other words, we observe that the increased analysis complexity does not degrade the performance on programs that do not need the new features. The loop specialization technique sometimes unrolls loops unnecessarily, which causes a modest slowdown for some benchmarks, suggesting that adjustments of the heuristics in Section 6.1 may be beneficial.

A possible threat to validity of our results is that the benchmarks in groups A, B, and C are not independent of each other: the different versions of jQuery naturally share some code, and group C is directly constructed from group B by removing unreachable code to be able to explore the relevant parts of the library. Nevertheless, these experiments demonstrate the effect of the analysis techniques and can be helpful for identifying opportunities for future work.

Another potential concern is that we have no proof of soundness, although this is no different from all existing work on static analysis for JavaScript web applications. (TAJS has known soundness bugs, but those we are aware of do not affect our conclusions.) In addition to the existing test suite in TAJS we have used its ability to detect dead code as a simple soundness check: If the analysis reaches a fixpoint where some code is reported as dead, the analysis is unsound if that code is reached in a concrete execution in a browser. This technique has caught a few subtle bugs in the modeling of the DOM compared with how modern browsers work.

We have performed a preliminary investigation of the unsuccessful cases (i.e. those where the analysis did not terminate within one minute), using the methodology from Section 4. Typically in the unsuccessful cases, unknown dynamic property reads (x[y] with an unknown y) lead to much spurious dataflow and consequently many spurious call edges, as exemplified in Sections 1 and 6. We have identified some of the major causes of these unknown dynamic property reads, which suggests the following three areas where additional analysis precision may be needed for successful analysis of more benchmarks. (1) The loop unrolling heuristic in Section 6.1 should unroll more loops that ultimately influence dynamic property reads. An example of this is a large loop that iterates over determinate strings, parses them as CSS selector strings, and stores the parse results as strings that are later used as keys in dynamic property accesses. (2) The control sensitivity for overloaded functions should be increased. Otherwise all the properties of the global object will be read in a dynamic property read for some uses of jQuery. Increased precision for type predicate functions, such as `isFunction`, is likely to help with this. (3) DOM elements and events should be treated with more precision, as some DOM object property values are used as keys in dynamic property reads and for deciding which overloaded function variants to use. An improved analysis could then avoid querying DOM elements using unknown dynamic property reads in places where the analysis should query the numeric properties of an internal cache object instead.

Our results are based on small jQuery applications only, so further studies are required to determine how indeterminacy in the application code affects analysis of the library. Although some JavaScript programs will surely remain beyond reach of sound static analysis, we conjecture that many real-world jQuery applications can be analyzed effectively by refining the techniques presented in this paper.

10. Conclusion

Previous work on static analysis for JavaScript web applications has shown that jQuery and related libraries are remarkably difficult to analyze efficiently, yet there is a substantial potential in creating effective analysis tools for such code. This could, for example, lead to better IDE support or tools that can perform optimizations or help programmers detect type-related errors during development.

We have shown how determinacy information can be inferred and exploited by a static analysis to enable analysis of a significantly larger part of the jQuery code base than previous tools have accomplished. Most importantly, we have demonstrated the decisive effect of combining selective context sensitivity with constant propagation and branch pruning. Each of the techniques we apply increases analysis precision, yet the consequent increased theoretical worst-case complexity does not manifest in practice in our experiments.

Although much work remains to be done, these results suggest that high-precision dataflow analysis is a promising approach to static analysis of real-world JavaScript code. Our investigation of the remaining challenges points toward more precise treatment of loops, overloaded functions, and DOM objects as potential topics of future work. Many variations of the heuristics proposed in Sections 5–7 can be conceived, which may be worthwhile to explore further.

Acknowledgments This work was supported by Google, IBM, and the Danish Research Council for Technology and Production.

References

[1] F. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1971.

[2] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proc. 19th European Conference on Object-Oriented Programming*, July 2005.

[3] G. Balakrishnan and T. W. Reps. Recency-abstraction for heap-allocated storage. In *Proc. 13th International Static Analysis Symposium*, August 2006.

[4] T. Ball and S. K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, June 2001.

[5] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proc. 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009.

[6] ECMA. ECMAScript Language Specification, 3rd edition, 2000. ECMA-262.

[7] A. Feldthaus and A. Møller. Semi-automatic rename refactoring for JavaScript. In *Proc. 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2013.

[8] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. In *Proc. 26th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2011.

[9] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proc. 35th International Conference on Software Engineering*, May 2013.

[10] S. Guarnieri and V. B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proc. 18th USENIX Security Symposium*, August 2009.

[11] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable JavaScript. In *Proc. 20th International Symposium on Software Testing and Analysis*. ACM, July 2011.

[12] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Proc. 18th International Conference on World Wide Web*. ACM, May 2009.

[13] B. Hackett and S. Guo. Fast and precise hybrid type inference for JavaScript. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2012.

[14] D. Jang and K.-M. Choe. Points-to analysis for JavaScript. In *Proc. 24th Annual ACM Symposium on Applied Computing, Programming Language Track*, March 2009.

[15] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium*, August 2009.

[16] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *Proc. 17th International Static Analysis Symposium*, September 2010.

[17] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proc. European Software Engineering Conference / ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2011.

[18] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proc. 21st International Symposium on Software Testing and Analysis*, July 2012.

[19] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977. Springer.

[20] V. Kashyap, J. Sarracino, J. Wagner, B. Wiedermann, and B. Hardekopf. Type refinement for static analysis of JavaScript. In *Proc. 9th Symposium on Dynamic Languages*, October 2013.

[21] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2013.

[22] B. S. Lerner, L. Elberty, J. Li, and S. Krishnamurthi. Combining form and function: Static types for JQuery programs. In *Proc. 27th European Conference on Object-Oriented Programming*, July 2013.

[23] F. Logozzo and H. Venter. RATA: Rapid atomic type analysis by abstract interpretation - application to JavaScript optimization. In *Proc. 19th International Conference on Compiler Construction*, March 2010.

[24] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proc. European Software Engineering Conference / ACM SIGSOFT Symposium on the Foundations of Software Engineering*, August 2013.

[25] M. Might and O. Shivers. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *Proc. 11th ACM SIGPLAN International Conference on Functional Programming*, September 2006.

[26] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1), 2005.

[27] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proc. 9th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1994.

[28] T. W. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1-2):206–263, 2005.

[29] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Dynamic determinacy analysis. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2013.

[30] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Proc. 4th International Symposium on Static Analysis*, September 1997.

[31] M. Sharir and A. Pnueli. Two approaches to interprocedural dataflow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[32] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[33] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *Proc. 26th European Conference on Object-Oriented Programming*, June 2012.

[34] P. Thiemann. Towards a type system for analyzing JavaScript programs. In *Proc. Programming Languages and Systems, 14th European Symposium on Programming*, April 2005.

[35] W3Techs. Usage of JavaScript libraries for websites, 2014. `http://w3techs.com/technologies/overview/javascript_library/all`.

[36] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 12(2):181–210, 1991.

[37] S. Wei and B. G. Ryder. Practical blended taint analysis for JavaScript. In *Proc. 22nd International Symposium on Software Testing and Analysis*, July 2013.

[38] B. Yankov et al. TypeScript type definition for jQuery, 2014. `https://github.com/borisyankov/DefinitelyTyped/blob/master/jquery/jquery.d.ts`.

[39] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proc. 20th International Conference on World Wide Web*, March/April 2011.