Remedying the Eval that Men Do

Simon Holm Jensen
Aarhus University, Denmark
simonhj@cs.au.dk

Peter A. Jonsson
Aarhus University, Denmark
pjonsson@cs.au.dk

Anders Møller
Aarhus University, Denmark
amoeller@cs.au.dk

ABSTRACT

A range of static analysis tools and techniques have been developed in recent years with the aim of helping JavaScript web application programmers produce code that is more robust, safe, and efficient. However, as shown in a previous large-scale study, many web applications use the JavaScript `eval` function to dynamically construct code from text strings in ways that obstruct existing static analyses. As a consequence, the analyses either fail to reason about the web applications or produce unsound or useless results.

We present an approach to soundly and automatically transform many common uses of `eval` into other language constructs to enable sound static analysis of web applications. By eliminating calls to `eval`, we expand the applicability of static analysis for JavaScript web applications in general.

The transformation we propose works by incorporating a refactoring technique into a dataflow analyzer. We report on our experimental results with a small collection of programming patterns extracted from popular web sites. Although there are inevitably cases where the transformation must give up, our technique succeeds in eliminating many nontrivial occurrences of `eval`.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Languages

Keywords

JavaScript, Refactoring, Static analysis

1. INTRODUCTION

The `eval` function and its variants in JavaScript allow dynamic construction of code from text strings. This can be useful for parsing JSON data[1], lazy loading of code[2], and execution of user code in web-based JavaScript IDEs[3]. Using `eval`, however, makes it difficult to statically reason about the behavior of the application code. Existing automated static analyses for JavaScript try to dodge this problem. They either forbid `eval` altogether [23, 1, 14, 19], handle only the simplest cases where the strings passed to `eval` are constants or assumed to contain JSON data [16, 15, 13], or simply ignore calls to `eval` thereby sacrificing precision and soundness [12]. Since JavaScript has limited encapsulation mechanisms, the dynamically constructed code can generally affect most of the application state, so ignoring calls to `eval` may have drastic consequences for the analysis quality.

The recommended best practice for web application developers is to avoid `eval`: *"The `eval` function is the most misused feature of JavaScript. Avoid it."* [5]. Nevertheless, the recent study "The Eval That Men Do" by Richards et al. has shown that `eval` is widely used [21]. Not only do a majority of the most popular web sites use `eval`, but in many cases they use it where simple alternatives exist, for example to access variables in the global scope or to access properties of objects. A likely explanation is poor understanding of the JavaScript language, in particular of its functional programming features that allow functions to be passed as arguments and of its unusual object model where each object is effectively a map from strings to values. Consequently, there is currently a mismatch between the capabilities of state-of-the-art static analysis tools for JavaScript and the JavaScript code that average programmers write.

Richards et al. also suggest that many of the uses of `eval` could be eliminated by rewriting the code, often improving both clarity and robustness as a side effect. They conclude that 83% of `eval` uses in their study could be rewritten to use less dynamic language features – however, they provide no automated way to perform these changes. Although it is often "obvious" to competent programmers how specific calls to `eval` can be eliminated manually, automating the transformation is not trivial. On the other hand, not all occurrences of `eval` can be eliminated with reasonable means; as an example, a call to `eval` that gets its input from an HTML text field could ultimately be eliminated by implementing a full JavaScript interpreter in JavaScript, which would hardly help static analysis tools reason about the code.

The goal of our work is to develop a sound, automated transformation technique for eliminating typical patterns of `eval` calls in JavaScript programs. The primary purpose is not to clean up messy code but rather to enable static analysis of programs that contain `eval`, for example for verification or bug detection. We therefore accept transformations that produce complex code as output as long as that code – unlike the input code that uses `eval` – is amenable to static analysis. We only permit transformations that preserve the behavior of the code because we want to apply sound static

[1]`http://www.json.org/js.html`
[2]`http://ajaxpatterns.org/On-Demand_Javascript`
[3]`http://tide4javascript.com/`

```
 1 function _var_exists(name) {
 2   // return true if var exists in "global" context,
 3   // false otherwise
 4   try {
 5     eval('var foo = ' + name + ';');
 6   }
 7   catch (e) {
 8     return false;
 9   }
10   return true;
11 }
12 var Namespace = {
13   // simple namespace support for classes
14   create: function(path) {
15     // create namespace for class
16     var container = null;
17     while (path.match(/^(\w+)\.?/)) {
18       var key = RegExp.$1;
19       path = path.replace(/^(\w+)\.?/, "");
20       if (!container) {
21         if (!_var_exists(key))
22           eval('window.' + key + ' = {};');
23         eval('container = ' + key + ';');
24       }
25       else {
26         if (!container[key]) container[key] = {};
27         container = container[key];
28       }
29     }
30   }
31 };
```
Figure 1: Example of `eval` taken from the Chrome Experiments program `canvas-cycle`.

analyses on the resulting code. In this way, eliminating `eval` can be viewed as a code refactoring challenge [7]. We want a tool to transform the program code without affecting its behavior, which requires an analysis to check certain preconditions and infer other information needed by the transformation. This apparently raises a chicken-and-egg problem: Before we can rewrite a given occurrence of `eval` we need to run a static analysis to infer the necessary information, but as discussed above we cannot in general perform static analysis of programs that use `eval`.

Another challenge is that the flexibility of `eval` makes apparently simple cases surprisingly difficult. For example, consider a rewrite rule that replaces a call `eval("`*S*`")` by *S* when *S* is a constant string consisting of syntactically correct JavaScript code. Such a rule is unsound; for example, *S* may contain variable and function declarations even when the call `eval("`*S*`")` occurs inside an expression, so the resulting code might not be syntactically correct, and moreover, variable declarations in *S* may conflict with variables in the surrounding code. Even finding the occurrences of calls to `eval` is nontrivial because programs may create aliases of the `eval` function. Some programs use such aliasing to exploit a subtlety of the language specification: Calling `eval` directly will cause the given code to be executed in the current scope, whereas calls via aliases use the global scope (or, before 5th edition of ECMAScript, cause an `EvalError` exception).

The example in Figure 1 demonstrates how `eval` can be used in practice. The code appears in the Chrome Experiments program `canvas-cycle`[4] and is part of a larger library that implements a class system in JavaScript, which does not support classes natively. This particular snippet implements a namespace mechanism for these classes.

[4]`http://www.chromeexperiments.com/detail/canvas-cycle/`

The example contains three calls to `eval`. The first on line 5 tests whether a given name exists in the global scope (although it only works if the name is not `"name"` or `"foo"`). This could have been accomplished without `eval`, for example by writing `name in window` since `window` refers to the global scope. The second call to `eval` on line 22 is used to assign to a dynamically computed property of the `window` object. This could have been achieved using `window[key]` to access the dynamically computed property. The last `eval` call on line 23 could be rewritten in a similar way. This example demonstrates that many calls to `eval` are in fact unnecessary and the same results could be achieved with other language constructs that are easier to reason about for a static analysis.

1.1 Contributions

The key idea of our approach is to eliminate `eval` calls soundly and automatically by incorporating refactoring into the fixpoint computation of a dataflow analyzer. We demonstrate this idea using the TAJS analysis [16, 17, 15] that performs a whole-program dataflow analysis for JavaScript web applications, but until now with poor support for `eval`. Whenever the analysis encounters dataflow into `eval`, a refactoring component is triggered for rewriting the call to equivalent JavaScript code without the `eval` call, and the analysis can proceed by analyzing the resulting code. When the analysis reaches its fixpoint, we have eliminated all reachable calls to `eval` and can output the resulting program. The success of this approach naturally depends on the power of the refactoring component and the information it can obtain from the underlying dataflow analysis – especially information about the strings that are passed to `eval`.

As an example, consider this fragment of JavaScript code used by Richards et al. for illustrating the power of `eval` [21]:

```
Point = function() {
  var x=0; var y=0;
  return function(o,f,v) {
    if (o=="r")
       return eval(f);
    else
       return eval(f+"="+v);
    }
}
```

A call `p = Point()` will return a closure that can be invoked as e.g. `p("w", "x", 42)` to write the value 42 to the local variable `x` or as `p("r", "x")` to read its current value. Let us focus on the second `eval` call site. Suppose that our dataflow analysis first encounters a call `p("w", "x", 42)`. Provided that the analysis can keep track of the flow of values, it can infer that `eval` is called with the argument `"x"+"="+42`, which reduces to `eval("x=42")`. This `eval` call can safely be rewritten to the assignment `x=42`, and the dataflow analysis can proceed by analyzing the effect of that assignment, which will likely have consequences to other parts of the program. If the analysis later encounters another call, for example `p("w", "y", 87)`, things become more complicated. Even if the analysis knows that the value of `f` is always a valid, non-reserved identifier name and `v` is always a number, and the local variables `x` and `y` are merely properties of a scope object, it is difficult to rewrite the `eval`'d assignment `f+"="+v` into an object property assignment because JavaScript does not provide a way to obtain a reference to the local scope object. Instead, we use context sensitive dataflow analysis to keep the two calls to p apart. Assuming that the analysis in this way finds out that the only possible values of f are "x" and "y", the code may safely be transformed into the following by conditionally specializing the eval calls accordingly:

```
Point = function() {
  var x=0; var y=0;
  return function(o,f,v) {
    if (o=="r")
       return f==="x" ? x : y;
    else
       return f==="x" ? x=v : y=v;
    }
}
```

Another example is the function get_server_option in the code for the web site scribd.com:

```
var get_server_option =
  function(name, default_value) {
  if (typeof Scribd.ServerOptions == 'undefined' ||
      eval('typeof Scribd.ServerOptions.' + name)
        == 'undefined')
    return default_value;
  return eval('Scribd.ServerOptions.' + name);
};
```

The dataflow analysis can find out that the value of name is always a valid identifier name by looking at the call sites, so the code can safely be rewritten to eliminate the calls to eval:

```
var get_server_option =
  function(name, default_value) {
  if (typeof Scribd.ServerOptions == 'undefined' ||
      typeof Scribd.ServerOptions[name]
        == 'undefined')
    return default_value;
  return Scribd.ServerOptions[name];
};
```

The transformations in these examples allow subsequent program analyses to reason about the code without having to worry about eval.

This paper explores the idea of incorporating eval refactoring into the dataflow analysis fixpoint computation and proposes a sequence of steps for developing the refactoring component and exploiting information provided by the dataflow analysis. In summary, the contributions of this paper are as follows.

- We describe a framework that soundly integrates refactoring of eval calls into a dataflow analyzer.
- Guided by a study of how eval is being used in practice, we instantiate our framework with different techniques for transforming typical calls to eval into equivalent JavaScript code without eval.
- We present results of an experimental evaluation with a prototype implementation. On 28 nontrivial programming patterns extracted from the Alexa top 500 web sites and from Chrome Experiments[5] containing a total of 44 calls to eval, our approach successfully eliminates 33 of the calls, which enables further use of static analysis on those applications and demonstrates that our approach is feasible. For the other call sites, we describe the challenges that remain for future work.

[5]http://www.chromeexperiments.com/

The remainder of this paper is organized as follows. Section 2 contains a study of calls to eval, slightly extending the work by Richards et al., to learn more about how eval is being used in practice. We present an overview of our transformation framework, the *Unevalizer*, in Section 3. We take the first step in Section 4 to eliminate a class of calls to eval where the arguments are constant strings and proceed with a number of improvements in Section 5 involving constant propagation, special treatment of strings that contain JSON data or identifiers, and context sensitive specialization to obtain more precise information about the strings that enter eval. In Section 6 we report on experiments performed using our prototype implementation on a small collection of JavaScript web applications that use eval and until now have been out of reach for static analysis tools.

Although our presentation focuses on the eval function, our technique also works for its cousins Function, setInterval, and setTimeout, and in principle for script code embedded in dynamically constructed HTML and CSS data. We target the 3rd edition of ECMAScript [6]. None of the web sites we have studied use the newer strict mode semantics in combination with eval.

The intended user of our code transformation tool is the JavaScript web application developer. This means that we can disregard "minification" and lazy code loading, which are often used before deployment to compress the code and divide it into small parts for faster loading, and we can assume that all relevant source files are available for analysis.

We strive toward transformations that preserve the program behavior: Given a program that uses eval, our tool either outputs a program with the same external behavior, but without eval, or the tool gives up and issues an explanation message. (Stating this formally and proving correctness is beyond the scope of this paper.) Since the main purpose of our work is to enable sound static analysis of programs that use eval, one may argue that we could loosen this requirement and permit non-behavior preserving transformations as long as they are sound with respect to the subsequent analysis. The advantage of our present approach is that the transformation of eval call becomes independent of the subsequent analysis of the transformed programs.

1.2 Related Work

Static analysis of JavaScript has been the focus of much work recently, and the eval function is widely recognized as being a challenging language construct.

Thiemann has suggested a type system for detecting suspicious type conversions [23], Anderson et al. have proposed a type inference algorithm for tracking object properties [1], Jang and Choe have presented a points-to analysis for a subset of JavaScript [14], and Logozzo and Venter have introduced an analysis technique that enables type specialization optimizations [19]. All these analyses are defined on subsets of JavaScript that do not include eval. The end result is that these analyses currently do not work for many real JavaScript programs.

In the Gatekeeper project, Guarnieri and Livshits mitigate the eval problem by providing a runtime checker that determines if a given JavaScript program falls into the safe subset [11]. Another approach, which is used in the Actarus security analysis tool by Guarnieri et al. [12], is to simply ignore the effects of eval, which makes analysis results unsound in the presence of eval calls.

Dynamically constructed code also presents unique challenges in security analyses that are performed on-the-fly whenever untrusted third-party code is loaded dynamically. Staged or incremental analysis [3, 10] handles the issue by generating security policies that are checked when code is loaded and added to the program using `eval`. In contrast, we disregard lazy code loading as discussed above, and our approach aims to eliminate `eval` calls by purely static dataflow analysis without runtime checks.

Some uses of `eval` follow common patterns that can be recognized and handled without needing a full analysis. The control flow analysis by Guha et al. recognizes loading of code [13], and our previous work uses similar techniques to rewrite uses of `eval` that simulate simple higher-order functions [15]. In the present work we aim to expand the scope of static analysis for JavaScript in general by transforming `eval` calls into other language constructs that can be handled by existing static analyzers.

We use TAJS [16, 17, 15] to drive the transformation of `eval` calls, but our approach is not inherently tied to TAJS. The general aim of TAJS is to detect likely programming errors related to mismatches of types and dataflow in JavaScript programs, for example to detect suspicious type coercions or function calls where the call expression may not evaluate to a function object. In brief, TAJS performs interprocedural flow-sensitive dataflow analysis with a complex abstract domain that soundly and in great detail models how objects, primitive values, expressions, and statements work in JavaScript according to the ECMAScript standard. Here, we do not use the results produced by TAJS when it analyzes a program; instead we exploit TAJS as a dataflow analysis infrastructure for exposing calls and arguments to `eval`. In previous work [15] we pointed at dynamically generated code as an important next step for static analysis of JavaScript web applications – we here take that step.

The ability to construct code from text at runtime is not limited to JavaScript. Most dynamic scripting languages include an `eval` construct. Furr et al. have presented an intermediate language to ease the task of making static analysis for Ruby [9]. Calls to `eval` are removed using dynamic profiling of the program during the transformation of Ruby programs into this intermediate form [8]. As for the comparison with staged or incremental analysis discussed above, the key difference with our work is that we aim for a sound and purely static approach. Interestingly, the experiments by Furr et al. suggest that `eval` is more commonly used for sophisticated metaprogramming in Ruby programs than Richards et al. have observed in JavaScript programs.

Other programming languages have more disciplined variants of `eval` than the one in JavaScript. As a case in point, `eval` in Scheme [22] works with S-expressions rather than text strings, which makes it easier to reason about the structure of the code being evaluated. Moreover, the code runs in an immutable environment, so it is safe to ignore `eval` calls in static analysis for Scheme, unlike JavaScript.

As mentioned above, our techniques can be viewed as a refactoring that transforms a program to a behaviorally equivalent one without dynamic code evaluation. Similar to the work we present here, Feldthaus et al. use static analysis as a foundation for describing and implementing refactorings of JavaScript programs [7]. One important difference is that we here perform the refactoring during the analysis, not after the analysis fixpoint is reached.

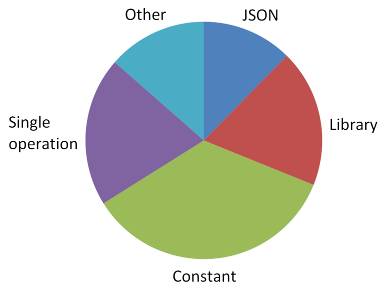

Figure 2: Classification of 17,665 `eval` call sites from Alexa top 10,000 web sites.

Knowledge about the contents of the strings that are passed to `eval` is obviously essential to be able to transform the `eval` calls to other code. As we show in the following sections, we have chosen a pragmatic approach that aims to cover the patterns that appear to be the most common in practice. This allows us to handle typical calls by focusing on relatively simple patterns of string concatenations. In principle, it would be possible to integrate more advanced string analysis algorithms, as introduced by Christensen et al. [2], but our study of how `eval` is used in practice suggests that our present approach is adequate in most cases.

2. EVAL IN PRACTICE

To guide the development and to be able to evaluate the quality of our code transformation system, we need a collection of representative example programs that use `eval` and show how it is used in practice. A useful starting point is the study by Richards et al. [21], which is based on execution traces of thousands of the most popular web sites according to Alexa[6]. Their study shows that more than half of the web sites use `eval`, which suggests that there are plenty of examples to choose from. We disregard dynamic code loading for the reason mentioned in Section 1.1, and JSON parsing can be treated separately with known techniques, which we describe in Section 5.2, so these uses of `eval` are less interesting to us. The Richards et al. study does not directly show how many of the web sites use `eval` for purposes other than dynamic code loading and JSON data parsing. Of the remaining uses of `eval`, calls where the argument is a constant string in the source code can also be considered as relatively easy cases for the transformation (we return to this category in Section 4).

To investigate this further, we examine the Alexa top 10,000 web sites. We find using the tools made available by Richards et al. that 6,465 of them use `eval`. Filtering out those that use `eval` for purposes other than dynamic code loading and JSON parsing gives us 3,378 URLs. If we further remove those where all calls to `eval` have constant arguments, only 2,589 URLs remain. This alone gives an interesting picture of the typical uses of `eval` that is not emphasized by Richards et al. [21]: Although `eval` is pervasive, we can expect that relatively few web sites (around 25%) use `eval` in ways that are truly challenging to reason about with static analysis.

A second observation is that the results of measuring the `eval` usage patterns are more useful to us if we count numbers of static call sites rather than numbers of runtime calls

[6]`http://www.alexa.com/topsites`

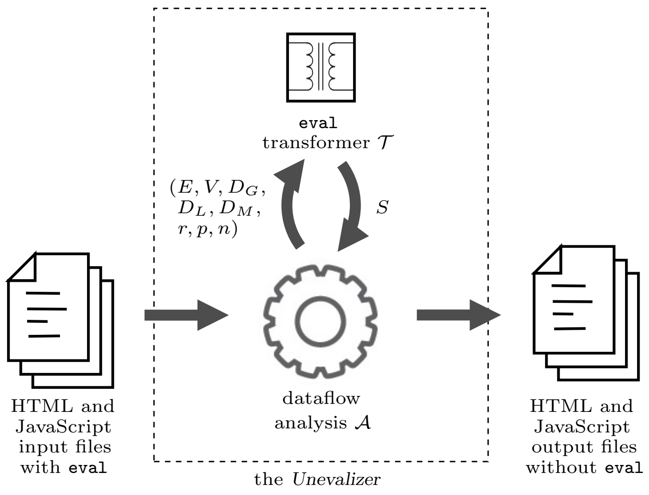
eval
transformer $\mathcal{T}$
$(E, V, D_G, D_L, D_M, r, p, n)$
S
HTML and JavaScript input files with eval
dataflow analysis $\mathcal{A}$
HTML and JavaScript output files without eval
the *Unevalizer*

Figure 3: Structure of the *Unevalizer*.

to `eval` as in the Richards et al. study. Many calls at runtime typically originate from the same call sites in the code, and for the purpose of developing techniques to transform source code to eliminate typical `eval` calls, we obtain more relevant information about the usage patterns by considering the static call site information. Of a total of 17,665 `eval` call sites, we find that 3,339 are used for loading library code, 6,228 have arguments that are constant strings (see Section 4), and 2,202 are used for parsing JSON data (see Section 5.2). Of the remaining call sites, 3,624 evaluate code strings that are single operations, such as property read/write operations, `typeof` type test expressions, or simple function/method calls. A few call sites, 141, fall into more than one of these categories. The distribution is shown in Figure 2. This suggests that a transformation technique that can handle constants, JSON, and single operations will cover a majority of the `eval` calls that programmers write.

3. THE UNEVALIZER FRAMEWORK

Figure 3 shows the structure of the *Unevalizer*. As input, it takes a JavaScript web application containing HTML and JavaScript files. It then transforms the application driven by a whole-program dataflow analysis and, if successful, outputs a semantically equivalent application that does not contain calls to `eval`.

The dataflow analysis $\mathcal{A}$ will abstractly trace all possible execution paths through the program and keep track of what data flows into what variables and functions. This process is based on the classical monotone framework [18] that maintains abstract states for all program points and abstract values for all expressions. Specifically, it models function objects using object labels where $\ell_{\texttt{eval}}$ is an object label describing the `eval` function that is defined in the ECMAScript core library. Our prototype implementation uses TAJS for the dataflow analysis. Whenever new dataflow is detected during the analysis at a function call site F`(`E`)`, where F and E are expressions, we look for calls to `eval`: If the abstract value provided by the analysis for F includes $\ell_{\texttt{eval}}$ then the transformation component $\mathcal{T}$ is triggered. Method calls are treated similarly as function calls, and we omit them here to simplify the presentation. We also ignore indirect calls via built-in native functions such as `call` and `apply`, which are fully supported by our analysis but rarely used in combination with `eval`.

The transformation component $\mathcal{T}$ is passed an 8-tuple $(E, V, D_G, D_L, D_M, r, p, n)$ with information from the analysis:

- E is the syntactic argument expression as it appears in the program code at the function call site.
- V is the abstract value of the argument expression E. This abstract value soundly approximates the code string to be evaluated.
- D_G and D_L are the sets of names of variable and function declarations in the global and local scope, respectively. This takes into account nesting of functions and properties of the global object. D_M is the set of names of built-in properties of the global object that may have been modified by the application code. We settle for sound approximations of these sets since JavaScript does not have ordinary lexical scope (due to `with` statements and dynamically constructed properties of the global object that are always in scope).
- r is a boolean flag that indicates whether the call appears syntactically as an expression where its return value is used (as in `x=eval(y)`) or as a statement on its own.
- p is a boolean flag that signals whether the `eval` call is direct or aliased, which controls its execution scope as mentioned in Section 1.
- n is a number that indicates the `eval` nesting depth, which is 0 for an `eval` call that occurs in the original source program, 1 for a call that appears in code generated by an `eval` call at nesting depth 0, etc.

This turns out to be sufficient information to perform the transformation in many common cases. Note that for a given call site we can statically determine E, r, p, and n from the syntax of the call and its context, whereas V, D_G, D_L, and D_M may vary during the analysis. We assume that the underlying dataflow analysis models possible string values of expressions using a finite-height lattice Str. We discuss specific choices of this lattice in Sections 4 and 5. On top of this, we give special treatment to argument expressions that are built from concatenations using the `+` operator, which is common in practice. As an example, for the call

```
eval("v"+i+"="+x)
```

the argument expression E is `"v"+i+"="+x` and its abstract value V is $v_1 \oplus v_2 \oplus v_3 \oplus v_4$ where each $v_1, \ldots, v_4 \in \mathsf{Str}$ are abstract values of the four constituents and $\oplus$ represents concatenation. Note that we do not require the underlying dataflow analysis to reason precisely about string concatenations, and the $\oplus$ operator is only used to model concatenations that appear literally in the argument expression E.

In response, $\mathcal{T}$ gives either

- a string S containing JavaScript code that is equivalent to the function call F`(`E`)` relative to the given context, or
- the special value $\lightning$ in case it is unable to transform the given `eval` call.

There will inevitably be situations where $\lightning$ is returned, for example if the value of E partly originates from the user via an HTML text field, as discussed in Section 1.

If T returns $\lightning$ then the *Unevalizer* is unable to transform the given program and aborts. If T returns successfully, the *Unevalizer* will incorporate S into the code base at the point of the function call and proceed with the analysis. In doing this, we must consider the possibility that $\ell_{\texttt{eval}}$ may not be the only value of F, in which case the analysis must process all the possible functions and join their respective abstract return states. Additionally, we must take into account the fact that E may evaluate to non-string values. Such arguments to `eval` are simply returned directly without string coercion according to the ECMAScript specification. Moreover, we must retain the original call F`(`E`)` in the code since more dataflow may appear later in the analysis, which triggers new invocations of $\mathcal{T}$. Consider the following example:

```
if (...)
  x = "f";
else
  x = "g";
...
eval(x + "()");
```

The first time dataflow arrives at the `eval` call site, it is possible that $\mathcal{A}$ has the information that the value of `x` is the string `"f"`, which could result in S becoming the code `f()`. However, $\mathcal{A}$ will later realize that `"g"` is also a possible value of `x`, and this may cause a different output from $\mathcal{T}$ replacing the old value of S.

As common in dataflow analysis using the monotone framework, the *Unevalizer* operates as a fixpoint computation that starts with the empty abstract states and empty abstract values everywhere and then applies monotone transfer functions iteratively until the least fixpoint is reached [18]. When $\mathcal{A}$ encounters a call to `eval`, that gets replaced by the code S, which $\mathcal{A}$ subsequently models as an abstract transformer $\widehat{S}$ relative to the abstract domain in use. This informally explains how we avoid the apparent chicken-and-egg problem we mentioned in Section 1: At each call site where $\ell_{\texttt{eval}}$ occurs, the corresponding values V, D_G, D_L, and D_M grow monotonically during the process. This requires the transformation component to be monotone in the following sense:

Property 1 (Monotonicity). *Let $C = (E, V, D_G, D_L, D_M, r, p, n)$ and $C' = (E, V', D'_G, D'_L, D_M, r, p, n)$ be inputs to $\mathcal{T}$ such that $V \sqsubseteq_{\mathsf{Value}} V'$, $D_G \subseteq D'_G$, $D_L \subseteq D'_L$, $D_M \subseteq D'_M$ where $\sqsubseteq_{\mathsf{Value}}$ is the partial order of abstract values in the dataflow analysis, and let S and S' denote the outputs from $\mathcal{T}$, that is, $S = \mathcal{T}(C)$ and $S' = \mathcal{T}(C')$. Let $\widehat{S}$ and $\widehat{S'}$ denote the corresponding abstract transformers with respect to the abstract domain used by $\mathcal{A}$. The transformation component $\mathcal{T}$ is* monotone *in the sense that $\widehat{S} \sqsubseteq_{\mathsf{Trans}} \widehat{S'}$ for any such two inputs C and C' where $\sqsubseteq_{\mathsf{Trans}}$ is the partial order of the abstract transformers.*

As the *Unevalizer* replaces calls to `eval` with other code that is analyzed subsequently, we must be careful with generated code that itself calls `eval`, although that is not common in practice. An example from `bild.de`:

```
eval("try { lFrame = eval(lf[i]) }catch(e){};");}
```

The `eval` nesting depth n gives us an easy way to ensure that the *Unevalizer* always terminates:

Property 2 (Convergence). *If $n > k$ for some bound k then $\mathcal{T}$ returns $\lightning$.*

The bound $k = 1$ suffices for all examples we have encountered.

We can now establish the meaning of correctness for the *Unevalizer* and the requirements to $\mathcal{A}$ and $\mathcal{T}$:

Property 3 (Correctness). *Assuming that*

- *the underlying dataflow analysis $\mathcal{A}$ is sound,*
- *for any input $(E, V, D_G, D_L, D_M, r, p, n)$, $\mathcal{T}$ outputs either $\lightning$ or a program fragment S that has the same external behavior as the call* `eval(`*E*`)` *in the context given by V, D_G, D_L, D_M, r, and p, and*
- *$\mathcal{T}$ satisfies Properties 1 and 2,*

the Unevalizer is guaranteed to output a program that has the same external behavior as the input, or report that it is unable to transform the input.

Upon completion, the *Unevalizer* outputs JavaScript and HTML files where all calls to `eval` have been eliminated. This allows the output to be be further analyzed by other analyses that do not work on programs that contain `eval`.

In the following two sections we describe our instantiations of the framework.

4. ELIMINATING CALLS TO EVAL WITH CONSTANT ARGUMENTS

We start by introducing techniques needed to remove calls to `eval` where the argument E is a constant string. Surprisingly many programs actually call `eval` with constant string arguments, as observed in Section 2. More importantly, this transformation is used as a stepping stone for Section 5 where we consider more general `eval` calls.

The task might appear trivial, but there are several issues to consider to ensure that the transformation is correct. A naive approach of simply "dropping the quotes" may yield a program with a different behavior. Consider the following hypothetical rewrite rule:

$$\texttt{eval("var x;")} \rightsquigarrow \texttt{var x;}$$

This rule might appear correct at a first glance, but consider the `eval` call below and the resulting program after applying the transformation:

```
var x = 2;                      var x = 2;
function f() {                  function f() {
  var y = x;                      var y = x;
  eval("var x;");       →         var x;
  return y;                       return y;
}                               }
f();                            f();
```

These two programs are not equivalent: the one on the right yields `undefined` rather than `2` since the global variable `x` is shadowed by the local with the same name.

In general, the following five issues must be considered when transforming `eval` calls with constant strings.

Statements. When the `eval` call occurs as an expression and E consists of statements rather than a single expression, the code must be reorganized using temporary variables to ensure a correct order of evaluation. For example,

```
x = a() * eval("b(); c();") * d();
```

can be translated into the following code:

```
var t1 = a();
b();
var t2 = c();
x = t1 * t2 * d();
```

This raises a subtle issue about generating fresh names, here t1 and t2. We pick names that are not in $D_G \cup D_L$, or return $\lightning$ in case that set contains all possible identifier names.

Declarations. Function and variable declarations in E can potentially clash with identifiers already in scope, as shown by the example in the beginning of this section. We simply let $\mathcal{T}$ return $\lightning$ if any new variable declarations in E are already in $D_G \cup D_L$.

Syntactic Validity. If the string passed to `eval` at runtime is not a syntactically valid program, a `SyntaxError` exception is thrown. This is easy for $\mathcal{T}$ to check when the string is a constant, simply by running a JavaScript parser. If the string is invalid, $\mathcal{T}$ returns $S =$ `throw new SyntaxError()`. The name `SyntaxError` may be shadowed by other declarations, so if `SyntaxError` $\in D_L \cup D_M$, we instead let $\mathcal{T}$ return $\lightning$. Although this is unlikely a problem in practice, it is necessary for soundness.

Return Value. The return value of `eval` is defined to be the value of the last so-called value yielding statement executed in the input string. Most statements have a value, however, a few such as the empty block and `var` statements do not. This means that the return value of an `eval` call cannot always be statically determined, even if the entire input string is a known constant. Consider for example this call:

```
eval("2;if (b) 3;")
```

Its return value is either 2 or 3 depending on the value of the b variable. Rather than trying to devise complex transformation rules to handle such cases, we choose a simple alternative that seems to suffice in practice: If the return value is not used, which $\mathcal{T}$ knows from the r flag, then there is no issue. Otherwise, we let $\mathcal{T}$ return $\lightning$ if it is ambiguous which statement will yield the return value. The string has already been parsed at this point, as discussed above, so checking for this kind of ambiguity is straightforward.

Scope. Another peculiar corner case in the ECMAScript standard is that the execution scope of dynamically evaluated code depends on whether `eval` is called directly or through an alias, which was the reason for introducing the p flag in Section 3. The following example uses an alias for `eval` to access a variable `x` in the global scope, even if the variable name `x` is shadowed by a local declaration:

```
var geval = eval;
geval("x = 5");
```

When the p flag is set to global scope execution, $\mathcal{T}$ needs to transform the code to ensure the proper binding of identifiers. At first, one may try to exploit the fact that the global object is a synonym for `window`, however, the `window` variable may itself be overwritten or shadowed by local declarations. A more robust way to get a reference to the global object is to evaluate the expression `(function () {return this;})()`, which we abbreviate as *global*. This is perhaps not pretty but it satisfies our requirement of being analyzable with, for example, TAJS. The call `geval("x = 5")` in the example above is then transformed into *global*`.x = 5`. Declarations in the global scope can be transformed similarly, for example `geval("function f(){...}")` becomes *global*`.f = function(){...}`.

One additional issue remains. Reading a nonexistent variable in JavaScript will throw a `ReferenceError`, but reading an absent property just yields the value `undefined`. If we change an identifier read operation naively into a property read operation, for example from `geval("x")` to *global*`.x`, the behavior changes if the identifier is undeclared. Instead we transform it into a conditional expression:

```
"x" in global ? global.x : throw new ReferenceError()
```

and check whether `ReferenceError` has been shadowed, as for `SyntaxError` earlier in the section.

5. MORE PRECISE ANALYSIS OF THE ARGUMENTS TO EVAL

Eliminating calls to `eval` with constant arguments as done in Section 4 handles the tip of the iceberg. We now suggest four pragmatic ways of building on top of the transformation described in the previous sections by more deeply exploiting the connection between the transformation component and the dataflow analysis.

5.1 Exploiting Constant Propagation

We obtain the first improvement using constant propagation, which the TAJS dataflow analysis already performs. Technically, the `Str` lattice mentioned in Section 3 contains all possible string constants and a top element $\top$ representing non-constant strings, and all transfer functions in TAJS are designed to perform constant folding.

The following example extracted from the web site `qq.com` demonstrates an `eval` call where simple constant propagation is enough to enable transformation:

```
var json = "<large constant string>";
...
eval("area="+json);
```

Consider also the following example from the Chrome Experiments program `canvas-sketch`[7] that uses `eval` to emulate higher-order functions:

```
if (vez.func instanceof Function) vez.func(texto);
else eval(vez.func + "(texto)");
```

It turns out that interprocedural constant propagation for this program is able to infer that `vez.func` is always a constant string. To handle an even larger class of `eval` calls, in Section 5.4 we present a way to boost the effect of constant propagation using code specialization.

5.2 Tracking JSON Strings

JSON is a standardized format for data exchange that is derived from the JavaScript syntax for objects, arrays, and primitive values [4]. It is designed such that JSON data can be parsed using `eval`, and many `eval` calls are used for this purpose as discussed in Section 2. Modern browsers have the function `JSON.parse` for parsing the JSON subset

[7]http://www.chromeexperiments.com/detail/canvas-sketch/

of JavaScript in a more safe and efficient manner. Many programs check whether the JSON object exists and, if not, fall back to calling eval for parsing JSON data.

The following pattern occurs in many web sites:

```
x = eval("(" + v + ")");
```

The wrapping forces v to be evaluated as an expression. If v is known to contain JSON data, this eval call can be translated as follows:

```
x = JSON.parse(v);
```

The benefit of this transformation is that JSON.parse, unlike eval, never has side-effects other than creating an object structure, so it can easily be modeled soundly in a static analysis.

We use the technique introduced in our earlier work [15] to find out which values contain JSON data: The Str lattice is augmented with a special abstract value JSONString that represents all strings that are valid JSON data. The transformation suggested above can then be applied whenever the abstract value V of E is, e.g., "(" $\oplus$ JSONString $\oplus$ ")".

Now, the problem is to detect when JSON data is created. This is easy for constant strings and for the function JSON.stringify that explicitly constructs JSON data, however, the most common source of JSON data is Ajax communication with the server. Since we cannot know what data the server produces by only analyzing the client-side of the web application, we choose to rely on user annotations in the JavaScript code to specify sources of JSON data, typically in Ajax response callbacks.

JSON data obtained using Ajax is in rare situations combined with other string values before being passed to eval. We leave it to future work to incorporate more elaborate string analysis [2] for reasoning about such cases.

5.3 Handling Other Non-Constant Strings

It is evident from Figure 2 that we need to handle other cases than constants and JSON strings. A common pattern is eval("foo."+x) that accesses a property of an object. This can be transformed into foo[x], but only if we can be certain that x evaluates to specific classes of values, such as numbers or strings that are valid identifier names. The transformation would be unsound if x has a value such as "f*2". This example suggests that we refine the Str lattice further: we introduce a new abstract value IdString representing all strings that are valid JavaScript identifiers. TAJS handles number values in a similar way as strings, so we here focus on the string values.

Related patterns such as eval("foo_"+x) and eval(x+ "_foo"), which also appear in widely used web applications, can be handled similarly. However, in the case of eval("foo_"+x) we can loosen the requirement on x. It suffices to know that x is a string that consists of characters that are valid in identifiers, excluding the initial character. We therefore extend Str with yet another abstract value IdPartsString representing such strings. As an example, the string "42" belongs to IdPartsString but not to IdString.

With these extensions, the *Unevalizer* can handle cases such as this one from canvas-cycle where $\mathcal{A}$ infers the abstract value IdString for the variable key:

```
eval('window.' + key + ' = {};');
```

In the following example from the web site zedo.com the abstract value of v0[i] is IdPartsString:

```
for(var i=0;i<v0.length;i++){
 if(eval("typeof(zflag_"+v0[i]+")!='undefined'")){ ...
```

When transforming calls such as eval("foo_"+x) that access identifiers with computed names we run into the problem described in Section 1.1 that JavaScript does not provide a general mechanism for accessing the current scope object, so we restrict ourselves to the cases where we are certain that the identifiers are not bound locally: if D_L contains names that in this case start with "foo_" then $\mathcal{T}$ returns $\lightning$.

5.4 Specialization and Context Sensitivity

By selectively exploiting context sensitivity of the dataflow analysis the *Unevalizer* can also handle many eval calls where the strings are not constant but can be traced to a finite number of constant sources. Consider the following representative example from the web site fiverr.com:

```
get_cookie = function (name) {
  var ca = document.cookie.split(';');
  for (var i = 0, l = ca.length; i < l; i++) {
    if (eval("ca[i].match(/\\b" + name + "=/)"))
      return decodeURIComponent(ca[i].split('=')[1]);
  }
  return '';
}
get_cookie('clicky_olark')
get_cookie('no_tracky')
get_cookie('_jsuid')
```

When the analysis enters get_cookie from the first call site, the name parameter will be bound to the constant string "clicky_olark". Constant propagation to the eval call will then enable transformation as in Section 4. When the analysis later encounters the second call to get_cookie, the name parameter would with a context insensitive analysis obtain the abstract value IdString, which would flow to the eval call and cause $\mathcal{T}$ to fail with $\lightning$. Instead, when name first flows to the eval call we mark that get_cookie shall be analyzed context sensitively with respect to the name parameter. This will ensure that the second and the third call to get_cookie with different arguments will be analyzed separately. As a result, the analysis will know that the only possible values of name at the eval call site are "clicky_olark", "no_tracky", and "_jsuid". This can be used to specialize the argument to eval and transform the eval call into the following expression:

```
name==="clicky_olark" ? ca[i].match(/\\bclicky_olark=/)
 : name==="no_tracky" ? ca[i].match(/\\bno_tracky=/)
 : ca[i].match(/\\b_jsuid=/)
```

This mechanism can in principle be taken a step further to handle situations where the eval call appears nested inside more function calls, similar to k-CFA or the use of call strings in interprocedural analysis [20], however, one level of selective context sensitivity seems to suffice in our setting.

6. EVALUATION

We have implemented the eval transformer $\mathcal{T}$ and use TAJS as the driving dataflow analysis, $\mathcal{A}$. The two are cleanly separated by an interface similar to the 8-tuple described in Section 3. Any program implementing this interface can in principle use the transformation component.

The implementation of $\mathcal{T}$ works by first converting the abstract value V to a concrete JavaScript program using placeholder identifiers for non-constant parts. This program is then parsed using an ordinary JavaScript parser, and the transformation is performed on the AST. If the transformation is successful, the output S is generated by pretty-printing the new program where the placeholders are replaced by the corresponding parts from E.

In this section we describe our experiences running the prototype on a benchmark collection. We will try to answer the following research questions about the *Unevalizer*.

Q1: Is the *Unevalizer* able to transform common usage patterns of `eval` calls?

Q2: To what extent are the individual techniques presented in Sections 4 and 5 useful in practice?

Q3: For call sites where the *Unevalizer* fails to find a valid transformation, can we suggest improvements that are likely to handle more cases?

6.1 Benchmarks

Our main source of benchmarks is the Alexa list[6] that we also used in Section 2. We focus on the most challenging cases of `eval`, which are the call sites that fall into the categories "other" or "single operation" described in Section 2. We exclude all web sites that do not have any instances of `eval` in these categories. Library loading is outside the scope of this work as discussed in Section 1.1, and the technique we use for JSON data in Section 5.2 has to some extent been covered before [15]. Applying these criteria on the Alexa top 500 list gives us 19 web sites.

Analyzing JavaScript web applications involves many other challenges than `eval`. Although TAJS is able to analyze many real applications [15], the 19 applications collected from the Alexa list are still beyond the current capabilities of TAJS because they are considerably larger than what we have run TAJS on previously. However, since the purpose of the present evaluation is not to test the quality of TAJS but how the *Unevalizer* performs, we choose to manually extract the parts of the web applications that involve calls to `eval` including the relevant dataflow. This exposes 25 interesting program slices, each containing one or more calls to `eval`.

Our previous experiments with TAJS considered programs from Chrome Experiments[5], which generally have more manageable sizes than the Alexa top 500 web sites. We have found 3 programs in Chrome Experiments that use `eval` in ways that satisfy the criteria mentioned above, and we include those programs unaltered without slicing.

The resulting 28 programs are listed in Table 1. For each of the sliced web sites, we list each program slice separately. The benchmark collection can be downloaded from `http://www.brics.dk/TAJS/unevalizer-benchmarks`.

6.2 Experiments

In this section we describe the experiments used to answer research questions Q1 and Q2. The last question, Q3, is discussed in Section 6.3.

Q1 is addressed by the column "Pass" in Table 1. The symbol ✓ indicates that the *Unevalizer* is able to successfully transform all `eval` call sites in the program, and × means that $\mathcal{T}$ returns $\lightning$ at some point during the fixpoint computation. We see that the *Unevalizer* is able to handle 19 out of 28 cases, corresponding to 33 out of 44 `eval` call sites.

We address Q2 with the three columns "ConstProp", "Identifier" and "Specialization" in Table 1. The numbers in those columns show how many call sites are handled by each of the three techniques presented in Sections 5.1, 5.3, and 5.4, respectively. Note that the specialization technique builds on top of constant propagation, but the numbers for "ConstProp" only include the cases that do not also require specialization.

Table 1: Experimental results. The first three programs are the ones from Chrome Experiments; the remaining ones are the sliced programs from the Alexa list. The columns "Call Sites" shows the number of `eval` calls, the next three columns show which techniques the *Unevalizer* uses to transform the calls, and the "Pass" column shows which programs are transformed successfully.

Site	Call Sites	ConstProp	Identifier	Specialization	Pass
`berts-breakdown`	1	-	-	-	×
`canvas-cycle`	1	-	-	1	✓
`canvas-sketch`	1	1	-	-	✓
`bild.de` (1)	1	-	1	-	✓
`bild.de` (2)	1	-	-	-	×
`conduit.com`	1	-	-	1	✓
`dailymotion.co.uk`	1	1	-	-	✓
`fiverr.com`	1	-	-	1	✓
`huffpost.com`	1	-	-	-	×
`imdb.com`	2	2	-	-	✓
`indiatimes.com`	2	2	-	-	✓
`myspace.com`	1	-	-	1	✓
`onet.pl` (1)	1	-	-	-	×
`onet.pl` (2)	1	-	-	-	×
`pconline.com.cn` (1)	1	-	-	1	✓
`pconline.com.cn` (2)	1	-	-	-	×
`rakuten.co.jp`	1	1	-	-	✓
`scribd.com`	2	-	-	2	✓
`sohu.com`	2	-	-	2	✓
`telegraph.co.uk` (1)	1	-	-	-	×
`telegraph.co.uk` (2)	2	-	2	-	✓
`washingtonpost.com`	1	-	-	-	×
`wp.pl`	1	1	-	-	✓
`xing.com`	3	-	-	-	×
`xunlei.com`	6	6	-	-	✓
`zedo.com` (1)	3	-	3	-	✓
`zedo.com` (2)	3	-	3	-	✓
`zedo.com` (3)	1	1	-	-	✓
Total	44	15	9	9	

We see that out of 44 call sites, constant propagation (Section 5.1) alone is enough to transform 15 `eval` call sites. Using identifier detection (Section 5.3) we eliminate 9 more call sites, and if we also add specialization (Section 5.4) 9 additional call sites are successfully transformed. These numbers suggest that all the techniques we have presented are useful in practice.

Example. An example of a successful transformation is `sohu.com`, which uses `eval` to create a form of dynamic dispatch based on property names in objects. The two `eval` calls appear in the same function `_SoAD_exec`:

```
function _SoAD_exec(o) {
 if (eval("typeof(" + o.t + "_main)") == "function")
   eval(o.t + "_main(o)");
}
```

The dataflow analysis determines from the call sites to the function `_SoAD_exec` that `o.t` has the abstract value IdString. Using the techniques in Sections 4 and 5.3, the sub-expression `o.t+"_main"` can be rewritten into a property read operation on the global object. To guard against potential clashes with identifiers in the local scope, the *Unevalizer* checks that no names in D_L have the suffix `"_main"`. The second `eval` call site is transformed in a similar manner. The resulting function looks as follows:

```
function _SoAD_exec(o) {
  if (typeof(
        (o.t + "_main") in global ?
            global[o.t + "_main"] :
            throw new ReferenceError())
       == "function")
      ((o.t + "_main") in global ?
            global[o.t + "_main"] :
            throw new ReferenceError())(o);
}
```

In this code *global* refers to the expression that returns the global object, as defined in Section 4. The conditional expressions ensure that a `ReferenceError` is thrown if the property is absent in the global object.

Threats to Validity. The fact that the *Unevalizer* successfully eliminates many nontrivial `eval` calls in some manually extracted program slices and a few medium size complete web applications obviously does not imply that all problems related to `eval` are now solved. Our manual slicing may be erroneous although we have strived to preserve all dataflow that is relevant for the `eval` call sites. Ideally, we would of course like to test our approach on a larger number of web applications and on the complete application code without slicing, but, as mentioned in Section 6.1, that requires a more scalable dataflow analysis than the current version of TAJS. With today's state-of-the-art analysis techniques for JavaScript, we see no better way of evaluating the *Unevalizer* than using the slicing approach. The programs included in the evaluation are all from real web sites and have been selected in a systematic and non-biased manner, following the criteria described in Section 6.1 that have exposed the most interesting cases of `eval`. We also point out that the *Unevalizer* can leverage from future improvements of TAJS or other dataflow analyses for JavaScript.

A second concern could be that the web sites from the Alexa list, which was also the foundation for Richards et al. [21], and the Chrome Experiments may not be representative for JavaScript web applications in general, however, we believe the programs included in the evaluation give a good indication of how `eval` is being used in practice.

6.3 Directions for Future Improvements

To answer Q3 we examine the cases where the *Unevalizer* fails to transform an `eval` call site. Overall we observe two reasons for failure: insufficient precision of the dataflow analysis on loop control structures (this accounts for 6 of the 11 failing `eval` call sites), and `eval` call sites where the argument is built from string concatenations that do not appear syntactically inside at the function call (4 cases).

Loops seem to cause a loss of precision that often hinders transformation. The following example from the web site `bild.de` demonstrates such a case:

```
for (var libName in $iTXT.js.loader) {
  currentLibName = libName;
  eval(libName + '_Load()');
}
```

The loop iterates over all the properties of an object, which is defined by a constant object literal elsewhere in the code. The property names do not match IdStrings, however, so the abstract value of `libName` becomes $\top$, which is insufficient to transform the `eval` call site. Applying loop unrolling in $\mathcal{A}$ to this example would enable better constant propagation, which could in turn enable transformation of the call site.

Recall from Section 3 that we give special treatment to string concatenations that appear syntactically in the `eval` argument expressions. This works well for the majority of our benchmarks, although the following example from `pconline.com.cn` shows a situation where it is inadequate:

```
function showIvyViaJs(locationId) {
  ...
  var _fconv = "ivymap[\'"+locationId+"\']";
  try {
    _f = eval(_fconv);
    ...
  } catch(e) {}
}
```

The string given to `eval` is created from concatenations, but not at the call site, and the abstract domain Str for string values in TAJS is not detailed enough to model the possible values of `_fconv` with sufficient precision. The abstract value V then becomes $\top$, which causes the *Unevalizer* to give up. One way to improve this would be to extend the constant propagation in $\mathcal{A}$ to propagate entire expressions. In the example, this could propagate the expression `"ivymap[\'"+locationId+"\']"` directly into the `eval` call, and then $\mathcal{T}$ would be able to handle it. Propagating expressions in a sound way is not trivial, however, as the order of evaluation must be preserved for certain operations.

Notice that both of the improvements suggested in this section could be implemented entirely inside $\mathcal{A}$ without modifying $\mathcal{T}$ or the general *Unevalizer* framework.

7. CONCLUSION

The `eval` function is in practice not as evil as some men claim. By incorporating an `eval` elimination refactoring into a dataflow analysis, we have demonstrated that it is often possible to eliminate calls to `eval` in a sound and automated manner and thereby enable static analysis of JavaScript programs that use `eval` in nontrivial ways. Although we base our proof-of-concept implementation, the *Unevalizer*, on the TAJS dataflow analysis infrastructure, our approach is not intimately tied to the inner workings of TAJS: any dataflow analysis that can safely provide the necessary information to the transformation component could be used. It is also possible to apply other analyses to the resulting program code, including many of those mentioned in Section 1.2.

Our experimental results suggest that the approach succeeds in eliminating typical uses of `eval`, but also that further improvements are likely possible within the framework.

Our future work will focus on the challenges related to `eval` calls that appear in loops and on extending constant propagation to handle entire expressions, as suggested in Section 6.3. Furthermore, now that many more JavaScript web applications are within range of static analysis, it becomes possible to explore new opportunities for improving other aspects of static analysis techniques for JavaScript.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

[1] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *Proc. 19th European Conference on Object-Oriented Programming*, volume 3586 of *LNCS*. Springer, July 2005.

[2] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium*, volume 2694 of *LNCS*. Springer, June 2003.

[3] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *Proc. 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009.

[4] Douglas Crockford. RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON). IETF. `http://tools.ietf.org/html/rfc4627`.

[5] Douglas Crockford. *JavaScript: The Good Parts.* O'Reilly, 2008.

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

[7] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for JavaScript. In *Proc. 26th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2011.

[8] Michael Furr, Jong hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proc. 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2009.

[9] Michael Furr, Jong hoon (David) An, Jeffrey S. Foster, and Michael Hicks. The Ruby intermediate language. In *Proc. 5th Symposium on Dynamic Languages.* ACM, October 2009.

[10] Salvatore Guarnieri and Benjamin Livshits. Gulfstream: Incremental static analysis for streaming JavaScript applications. In *Proc. USENIX Conference on Web Application Development*, June 2010.

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

[12] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable JavaScript. In *Proc. 20th International Symposium on Software Testing and Analysis.* ACM, July 2011.

[13] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for Ajax intrusion detection. In *Proc. 18th International Conference on World Wide Web.* ACM, May 2009.

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

[15] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proc. 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2011.

[16] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium*, volume 5673 of *LNCS*. Springer, August 2009.

[17] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *Proc. 17th International Static Analysis Symposium*, volume 6337 of *LNCS*. Springer, September 2010.

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

[19] Francesco Logozzo and Herman Venter. RATA: Rapid atomic type analysis by abstract interpretation - application to JavaScript optimization. In *Proc. 19th International Conference on Compiler Construction*, volume 6011 of *LNCS*. Springer, March 2010.

[20] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, October 1999.

[21] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do - a large-scale study of the use of eval in JavaScript applications. In *Proc. 25th European Conference on Object-Oriented Programming*, volume 6813 of *LNCS*. Springer, July 2011.

[22] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten, editors. *Revised^6 Report of the Algorithmic Language Scheme – Standard Libraries.* `http://www.r6rs.org/`, September 2007.

[23] Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *Proc. Programming Languages and Systems, 14th European Symposium on Programming*, volume 3444 of *LNCS*. Springer, April 2005.