Eval Begone!

Semi-Automated Removal of Eval from JavaScript Programs

Fadi Meawad Gregor Richards Floréal Morandat Jan Vitek

Purdue University

Abstract

Eval endows JavaScript developers with great power. It allows developers and end-users, by turning text into executable code, to seamlessly extend and customize the behavior of deployed applications as they are running. With great power comes great responsibility, though not in our experience. In previous work we demonstrated through a large corpus study that programmers wield that power in rather irresponsible and arbitrary ways. We showed that most calls to eval fall into a small number of very predictable patterns. We argued that those patterns could easily be recognized by an automated algorithm and that they could almost always be replaced with safer JavaScript idioms. In this paper we set out to validate our claim by designing and implementing a tool, which we call Evalorizer, that can assist programmers in getting rid of their unneeded evals. We use the tool to remove eval from a real-world website and validated our approach over logs taken from the top 100 websites with a success rate over 97% under an open world assumption.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques—Program editors; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering

General Terms Languages

Keywords Dynamic Languages, JavaScript, Reflection, Dynamic Analysis

1. Introduction

JavaScript's eval function gives programmers the ability to run JavaScript code generated at runtime. This gives programmers extraordinary flexibility, but at the cost of understandability, efficiency and safety. As strings passed to eval may come from any source, including computation, user input or another website, and as eval is capable of performing any task[1], it can serve as a "black hole" both for analysis and for maintenance; to understand its behavior, one must know every *potential* argument. Consider the following JavaScript expression:

```
eval ( x )
```

Depending on the value bound to variable x, the state of any heap-allocated mutable value and the bindings of local variables in scope can be modified as a side effect of evaluating this statement. While some languages can enforce some modicum of data abstraction, JavaScript has very little in the way of encapsulation mechanisms. The impact of an eval can span over the entire heap.

The existence of eval is a quandary for those wishing to perform static analyses on JavaScript code or enforce any kind of semantic invariants. With an unknown string, eval has no locality guarantees, no time or memory bounds, not even a termination guarantee. Since the strings frequently come from outside sources and are as such completely unknown, static analyses are forced to assume the worst, losing all potential gains from the analysis, and dynamic analyses are at best forced to reevaluate every time an eval is encountered. It is common for researchers to simply ignore it [1, 2, 12, 23], claim it is very rare [9], assume its use is innocuous [10], or acknowledge its problems but simply produce a warning when it is used [13], resulting in unsound or even unsafe results. In security literature, in particular, eval is viewed as a serious threat [22], and it is frequently forbidden [16], filtered [6] or wrapped [6], but all of these solutions imply a flexibility or speed penalty.

Most dynamic languages have an eval function or similar feature, and in many it is considered harmful, though not all. The R programming language is an example where eval is a key mechanism for language extensibility [17]. In previous work [21], we showed that eval is ubiquitous in the largest and most popular websites on the Internet; we speculated that over

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

[1] JavaScript provides several other such functions, such as setTimeout and Function, but we focus on eval for much of our discussion.

75% of evals could be replaced by other mechanisms within JavaScript. In the present paper, we set out to validate that claim and end up demonstrating that over *95% of the evals invocations in real websites can be replaced with less general mechanisms* with very little work on the programmer's part. Our working hypothesis is that no more than 5% of all evals are actually needed.

Examining the most common use cases of eval, we observe that programmers decide to use eval for one of the following reasons: they use eval to parse JSON and malformed-JSON. For parsing JSON, until recently some browsers did not have native JSON parsers, but this is no longer the case. Current native JSON parsers does not handle malformed-JSON. Others use eval to access or modify properties based on user input. It might require some small parsing to handle the input without using eval. And the last category of users is the one executing code coming from third party with no prior knowledge of its shape. Most modern browsers have native JSON parsers that is usually faster than eval for a JSON string, and it only accepts JSON strings, thus improving the security. Parsing user input to extract the correct expression might be hard to program manually, but once eval is used we loose the possibility of using static analysis as well as suffer the risk of code injection. When running third party code, we have noticed that it is usually the same or at least has the same shape. But the user would not be able to figure out what kind of strings are being executed and how to write a parser for them.

We see three possible roads to an eval-free Internet: *prohibition, prevention* and *migration.* Prohibition is conceptually the simplest. If calls to eval are simply disallowed, the problem is no more, but this comes at a cost in expressive power. Eval occasionally is the only practical way to achieve a certain degree of customizability in the behavior of a website, so forbidding it would reduce the expressive power of JavaScript. In other cases, eval is a way to delay design and implementation decisions, thus allowing developers to deploy applications faster. Again, losing that would decrease the usefulness of the language for rapid development. The second path is to prevent eval by proving, ahead of time, that they are not needed. This can be done through static program analysis techniques which construct approximate models of the program and can determine which strings flow into a particular eval call site. Indeed, there are many constant and quasi-constant strings that are passed in as argument to eval. A static analysis could be coupled with a compiler optimization that compiles the evals to equivalent code. This could be as simple as removing the quotes around a piece of text and splicing in the program where the eval used to be. Unfortunately there are inherent limits to static analysis approaches; they are unable to guess, for instance, what a user will type. In our experience, over 40% of strings passed to eval are, at least in part, generated by components outside of JavaScript code, such as the user, browser or server. We also observed that many of the most gnarly evals take their argument from outside the JavaScript program. This leaves migration as the third path. This is a path where we attempt wean users off their addiction to eval by showing them how to rewrite their code without it. We recognize that eval can be useful either because the program actually requires the flexibility that it brings to the table or because it is a handy crutch during an early phase of the program's life, but argue that in most cases, there is a point where the same functionality can be achieved without it.

We propose a simple dynamic approach to get rid of eval. The technique we present in this paper detects how eval is used through dynamic analysis of calls to eval. It categorizes the strings passed as argument to each call site of eval and proposes generic replacements that do not involve calls to eval. We realize our proposal in a tool which we name the Evalorizer. This tool aids in the evolution of eval-utilizing code to eval-free code by presenting replacements that fit the real use of each call site. This allows programmers to use eval in the development phase, when its flexibility may be most beneficial, then to gradually remove it in preference of simpler, safer and more readable solutions. We use a grammar inference algorithm to determine the used patterns of any given eval call site as a restricted subset of JavaScript's grammar, then generate succinct code that will handle all the same patterns, but with greater constraint. Additionally to aiding developers in removing evals, Evalorizer is also capable of *dynamic* eval removal. This technique can be used for verification, for measurement, or to perform analyses on otherwise hostile programs without intervention of the original developers. Though we focus on JavaScript, eval is certainly not unique to that language. Our implementation is specific to JavaScript, but our techniques are not, and are applicable to any language which provides a function similar to eval as well as other, safer reflective capabilities.

We evaluate the benefits of our approach by successfully migrating five real world websites. Furthermore, we use data obtained from the 100 top websites and evaluate the quality of our inference algorithm. We also measure the runtime performance impact of our technique.

Our tools and data are freely available at:

`http://sss.cs.purdue.edu/projects/dynjs`

2. Use case

We motivate Evalorizer with a typical use case. Assume that a web programmer, finding that the existing calls to eval on `CNN.com` inhibit maintainability, wants to determine if some of those calls can be removed. The programmer does not have intimate knowledge of all of the components of this site. The JavaScript frontend maybe easily understood, but the part that communicates with server components and third-party code is obscure. Because the argument strings of eval are unconstrained and may have come from these sources, the programmer does not know what these calls are doing or even if they are actually useful. Some of these calls may be needed,

but it isn't clear from looking at the code. CNN.com also uses code from a third party domain ATDMT.com that our developer does not trust entirely. To integrate both websites, she was instructed by ATDMT.com's representative to eval the code they are providing, but a safer and more predictable mechanism would be preferable.

Once launched, Evalorizer acts as an HTTP proxy, interposing on the traffic between the browser and the server. There is no specific requirement on where it has to be run, as it does not integrate into the server software, so the developer's own machine is suitable. The programmer may use any web browser, so long as it is configured to proxy all relevant traffic through Evalorizer. Fig. 1 illustrates this setup, as well as the communication between the different components of the system over time for a sample interaction.

Browsing the website is the only task left to the programmer. Ideally she should use as much of the site as is practical, to exercise every potential eval in the code; the more pages she browses and the more evals are triggered, the more accurate the results will be. Once the browsing session is complete, Evalorizer returns a log of all evals encountered in the code. For each call site, the user is provided with the arguments, the file name where it was located, and the absolute location within this file.

Our user visits the CNN.com homepage. The request is sent to the proxy ($\rightarrow 1$) that in turn forwards the request to the CNN.com server ($\rightarrow 2$). The server returns the homepage to the proxy ($\rightarrow 3$). The homepage had, in our example, two eval call sites. One was in the HTML page in an embedded script tag, and the other was from an external JavaScript file. The call sites are:

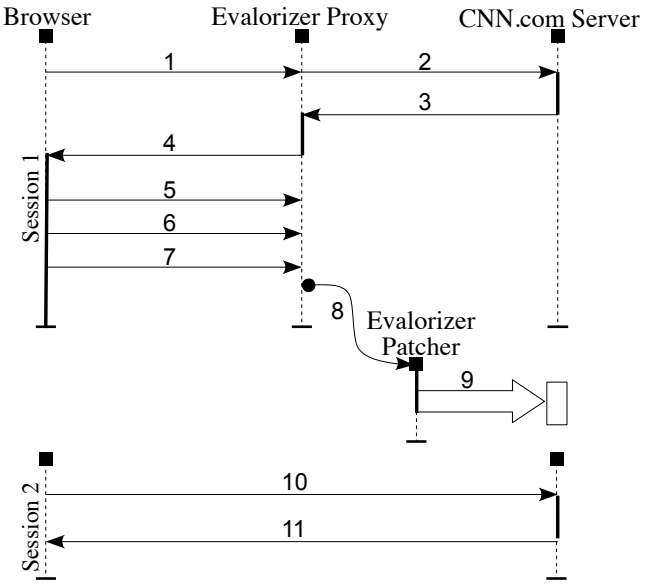
	filename	line	eval call site
1	index.html	105	t = eval(response);
2	lib1.js	1402	p = eval("window."+ x);

Inspecting the eval call sites or even its surrounding code does not provide any intuition on what eval is performing and what could be a replacement code. In the first call site, it just evaluates the variable response, that is obtained from an asynchronous call back. Performing post flow analysis for the left-hand-side t may give some intuition on how the return value of eval is used, but its side effects (including changes to the lexical environment) will remain obscure. We contend that the second eval call site, despite that part of the argument is a string literal, is exactly as obscure as the first one.

The proxy logs the call site information for future use, called the *static log*. Then it rewrites the eval call sites by invoking a function that will send the eval argument to the proxy, then call eval. The rewriting for our example is:

```
t = (_ev1=(response),logEval(_ev1, 1),eval(_ev1));
p = (_ev2 = ("window." + x),logEval(_ev2, 2),eval(_ev2));
```

logEval is a function that sends an asynchronous message to the proxy with the argument information; the first parameter



On the first session, the browser requests a page from the proxy ($\rightarrow 1$), that in turn forwards the request to the CNN.com server ($\rightarrow 2$). After the server sends the requested page to the proxy ($\rightarrow 3$), the later instruments the page and sends it to the browser ($\rightarrow 4$). When an eval is invoked, the browser sends a message to the proxy ($\rightarrow 5, 6, 7$). After the first session is complete, the patcher will acquire the logs form the proxy ($\rightarrow 8$), and creates a patch that can be used to update the server ($\rightarrow 9$). After patching, the browser communicates with the server directly ($\rightarrow 10, 11$).

Figure 1. A typical execution of Evalorizer.

is the eval argument, and the second is an identifier that maps the invocation to the call site. The proxy then sends the updated files to the browser ($\rightarrow 4$).

While the page was loading on the browser, the second eval has been invoked twice, with the arguments window.width and window.height. With some interaction with the page, for example by expanding a collapsible , the first call site is invoked with the argument

```
({"type" : "news",
  "value" : [{"order" : 2, "name" : "news1",
              "text":"This event has happened"}, .... ]})
```

For each invocation, the logEval ($\rightarrow 5, 6, 7$) function is called, and therefore the proxy now has a log of the invocations, called the *dynamic log*. She should attempt to visit every part of the page as well as visiting as many possible pages. The programmer can inspect the proxy logs at any time, and it will give her the number of invocations for each eval call site if any. Five invocations is usually sufficient to discover the exact functionality of the call site.

After this exercise is complete, the user will run the second component of our tool, the patcher. It uses both the static and dynamic logs ($\rightarrow 8$), and generates a patching script. The script depends on the policy used, whether you have full

confidence of the replacement, or not, and whether you want to inspect possible hacking attempts. A simplified version of the generated script for both call sites is (line numbers omitted):

```
--- index.html
-t = eval(response);
+t = JSON.parse(response.substring(1,
    response.length-1));
--- lib1.js
-p = eval("window." + x);
+p = window[x];
```

The programmer now should apply the patch to his original code ($\rightarrow 9$), remove the proxy configuration ($\rightarrow 10, 11$), and start using the website eval-free!

3. The State of the Eval

This section reviews the semantics and use of eval. JavaScript [7] is supported by all major web browsers and is an imperative, object-oriented language with Java-like syntax and a prototype-based object system. An object is a set of properties that behaves like a mutable map from strings to values. In JavaScript, eval is a function defined in the global scope. When invoked with a single string argument, it parses and executes the argument. It returns the result of the last evaluated expression. eval can be invoked in two ways: called directly, the eval'd code has access to all variables lexically in scope, when called through an alias, the eval'd code executes in the global scope [7]. While eval is powerful, its use is often unnecessary in practice. Consider,

eval("r.m_" + input)

For likely values of input, the above code could be implemented as

r["m_" + input]

Rather than invoking the full power of eval, indexing r returns the desired property with fewer surprises.

In previous work we identified patterns in the 10 K most popular web sites [21]. Some are industry best practices, but most result from poor understanding of the language, repetition of old mistakes, or adapting to browser bugs. These patterns can be detected by a simple syntactic check:

JSON	A JSON string.	{ "x": 1 }
JSONP	A use of JSON.	foo({"x":1})
Library	Function definitions.	function(){...}
Read	Read of a property.	x.foo
Assign	Assignment.	x.foo = 3
Typeof	Type test expression.	typeof(x)!="undefined"
Try	Trivial try-catch block.	try{ x=3; } catch{}
Call	Simple call.	x.foo(1,2,3)
Empty	Blank string.	""

JSON and JSONP are strings in JSON [7], a literal notation used for data interchange. JSONP pattern is often used for load balancing requests across domains. JSON parsing is now supported by browsers natively. The Library pattern captures uses of eval for loading code dynamically. The Read and Assign patterns capture access and assignment to local variables and object properties. The Call pattern captures function calls. The Typeof, Try and Empty categories are pathological cases with little raison d'être. We have found that eval call sites are quite consistent with respect to the pattern of the string argument. Across all of our data sets, we observed only 399 eval call sites (1.4% of all call sites) with strings in multiple pattern categories. Many of these "polymorphic" cases were clearly a single centralized eval used from many branches and for many purposes.

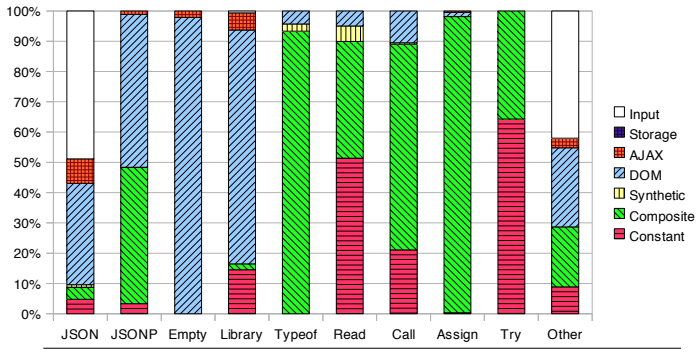


Figure 2. Provenance by Pattern. Distribution of string provenances across eval categories in each data set. X axis is the pattern that string falls into, Y axis is proportion of provenance in that category.

The eval strings come from different parts of the program. We have identified seven common provenances.

Constant	Appears in the source code.
Composite	Concatenated from constants and primitives.
Synthetic	Constant in a nested eval.
DOM	Obtained from DOM or native calls.
AJAX	Data retrieved from an AJAX call.
Cookies	Retrieved from a cookie or persistent storage.
Input	Entered by a user into form elements.

Fig. 2 shows the proportion of the different provenances by pattern. What is striking is that some patterns such as JSON (and the uncategorized grab bag Other) have close to 50% of their arguments coming from input. These numbers are relevant as they provide a natural upper bound on the ability of static analysis to infer eval behavior.

4. Translating Eval calls

The primary purpose of Evalorizer is to replace eval call sites with safer JavaScript idioms. Our approach is to collect eval call strings at every call site and infer a classifier sufficient to match all encountered strings. This concept was chosen as a generalization of our previous work [21], in which we showed that most eval strings fit into simple categories, and most eval call sites only receive strings in a single category.

Consider the typical unnecessary use of an eval call site fitting into the Read pattern, shown earlier:

```
var p = eval("window." + x);
```

Intuitively, this could be replaced by the following code, using JavaScript's map operator:

```
var p = window[x];
```

However, this intuition is built on an assumption about the value of x. Namely, that x is a string, and a valid JavaScript identifier. Without having actually seen the values of x, our naïve replacement may break the semantics of this eval call. This holds even for broken code; changing the way an error is reported may disrupt a deployed application.

Therefore, we generate a recognizer for those eval strings which are actually used at this eval call site. We generate this recognizer by collecting eval strings, parsing them as JavaScript, and finding their commonalities. Once the recognizer is made, we generate code to match these strings, and perform the same behavior. The generated code is constrained to accept, at a minimum, only those eval strings which have actually been seen; in practice it accepts more. These processes are detailed in this section.

4.1 Patterns vs. general classifiers

In our previous work, an eval taxonomy was proposed, and recurring patterns [19, 21] were discovered. In modern JavaScript, nearly none of these patterns are good uses of eval; the Library and JSONP patterns may be justifiable, as they may in certain conditions have no standard JavaScript replacement. The foundation of this work comes from the following key observation from the previous work: 98.7% of examined call sites have only one pattern as an argument. In other words, the overwhelming majority of eval call sites perform the same kind of task each time they are executed. The exact string passed in may differ, but it belongs to the same pattern.

Detecting patterns is unfortunately not enough to replace eval call sites. Indeed, even if an argument to a given eval call belongs consistently to the same pattern, the actual value may change considerably from one call to another. For instance eval(x) may receive the string "foo()" then "bar()" as argument. Both of them belong to the pattern Call, but actually these two calls are quite different, so a replacement for all eval call sites in the Call pattern would need to be quite general. Conversely, the call site may receive the strings "document.appendChild(a)" and "document.appendChild(b)", in which case our more general Call replacement would be both overly general and inefficient. By classifying and creating recognizers tailored to each eval call site, we can both avoid unnecessary code and provide more specific details about how each site is used.

4.2 Classifying arguments

Eval call sites are instrumented to record their arguments, and those arguments are collected and analyzed. This data is then used to translate the general call site into JavaScript code which accepts the same strings and performs the same actions with them, but does not accept unexpected and potentially malicious input.

Although static analysis may be an attractive alternative to find these classifications, we argue that static analysis would be forced to overlook many evals, as indeed many of the values taking part of the computation come from external websites, user input or from the DOM (Fig. 2). For this reason, we choose to use a dynamic analysis. Every time a new argument is recorded by Evalorizer, the system will parse the string and build an abstract syntax tree (AST) for this expression. Every time a call site encounters a distinct new AST, it is merged with previously seen trees. This merged tree is used as a recognizer, which represents a highly-restrictive subset of the JavaScript language. Many strategies may be considered for merging these trees, so long as they encode in some way the variable part of the strings. We will discuss in the following subsections the pros and the cons of some of these strategies.

Additionally to AST nodes, the trees may have choice nodes and generalization nodes. Choice nodes are a simple disjunction between two or more subtrees, generalization nodes will be described later. The language recognized by a tree with no choice or generalization nodes is a language of size one; with no options, only a single string is accepted. The language recognized by a tree with choice nodes but no generalizations is finite and regular. Generalizations match particular, chosen subsets of the JavaScript language, and allow a recognizer's language to be infinite and, if necessary, non-regular. We describe first the creation of choice nodes through merging, then the process of generalization.

To describe the behavior of each merging strategy, we will again use the following eval call site as an example:

```
var p = eval("window." + x);
```

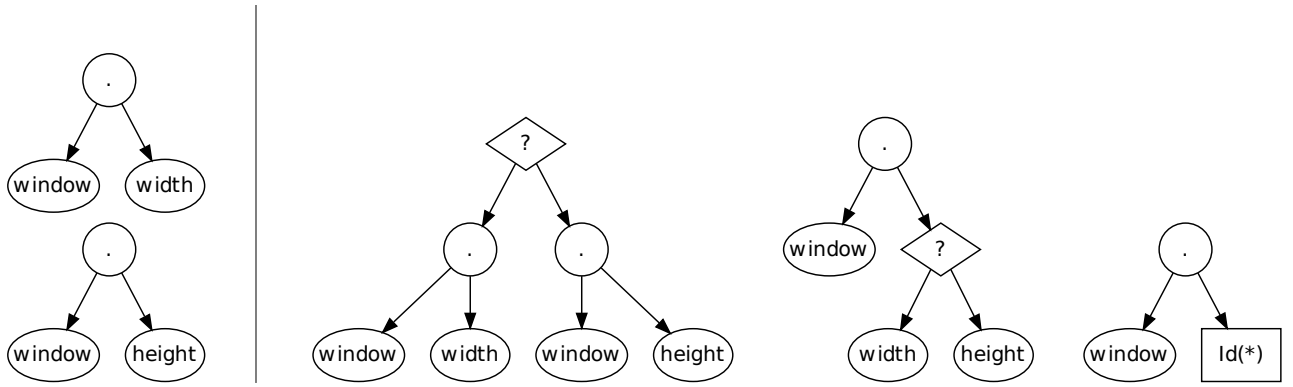
We will consider the simple case where x is always either "width" or "height", and so the strings passed to eval are "window.width" and "window.height". The ASTs for these expressions are shown in Fig. 3.

4.2.1 Choice

In order to accept more than one string, the recognizer must contain either choice nodes or generalizations. Choice nodes are created by merging the trees generated by multiple invocations of the same eval call site. There are several methods by which merging may be performed.

Constants. In the simplest case that the site is evaluated with the same string in all circumstances, the trees to be merged are identical, so the merging process yields the input tree. The AST itself acts as a recognizer. This AST will only recognize the exact string that was seen during analysis[2].

[2] As the recognition is based on the AST, whitespace change in the string are accepted as well



(a) Example trees. (b) Unbounded repetition. (c) Alternative nodes. (d) Member access generalization.

Figure 3. Example of merging the running window.width-vs-window.height example.

Unbounded repetition. The easiest way to merge two different trees is to consider them as outright alternatives (Fig. 3b), joining both trees at a single choice node. To match a new string against this recognizer, one simply needs to check it against each tree. This is the simplest, most straightforward approach. Unfortunately, because the number of alternatives are unbounded, this leads to an enormously inefficient recognizer; we essentially must check each string against every string that has ever been seen. This would lead to a size explosion in our generated code, which is unacceptable as in JavaScript the code size is considered essential. This technique has been applied for similar purposes in other settings [8]. For our simple example, this generates the tree shown in Fig. 3b, which has clear redundancies.

Alternative nodes. Instead of considering the trees themselves as full alternatives, they can be merged from top down to some differing point. When a node does not perfectly match, an *alternative* (or choice) node is inserted as a parent of these branches (Fig. 3c). The recognizer for this tree needs only to diverge when it encounters the nested choice node. This proposal reduces most of the drawbacks of previous option since common parts of trees are shared. However, in the worst case it is still unbounded. Our previous work [21] demonstrated that in most cases a single call site will evaluate strings of a single, simple pattern [21], thus code explosion is unlikely. For our simple example, this generates the tree shown in Fig. 3c. This recognizer accepts all of the example inputs, but no others.

A recognizer, using any of the last 3 techniques, can be built for any language, as long as it has an AST representation.

4.2.2 Generalization

Choice nodes allow us to capture a finite number of strings. In practice, this has two obvious problems: as the number of inputs becomes large, the number of choices at any choice node will also grow, and it does not adapt in any way to strings not seen in analysis. Generalization is the process of replacing subtrees of a recognizer's tree with broader parsers for chosen subsets of the JavaScript language. In principle, *any* subtree could be replaced with a general parser; in practice, this doesn't work. We could, for example, replace all trees by a parser for the entirety of JavaScript, but then the purpose of Evalorizer would be defeated: once code had been recognized, the only way to evaluate it would be with eval itself. As such, we generalize only those subsets of JavaScript for which there is a clear replacement for eval which accepts the same subset of JavaScript. Although the particular generalizations and replacements are therefore JavaScript-specific, the technique is usable in any language with reflective features.

Member expressions. The dot operator in JavaScript can generally be replaced by the map operator, and the map operator accepts a string.

$$\text{eval("window."+ x)} \longrightarrow \text{window[x]}$$

Because this replacement strategy is available, we may replace the right hand side of dot operators by generalizations which accept all JavaScript identifiers. Our simple example fits this case, so it may be replaced by the tree presented in Fig. 3d. It is important to note that this generalized tree accepts a much larger (in fact, infinitely larger) subset of JavaScript than the previous tree (Fig. 3c). Although our approach is to prefer generalization wherever possible, this change in recognized language may have security implications.

Literal primitives. Numeric and string literals embedded in JavaScript strings may be evaluated by Number and JSON.parse[3], respectively.

$$\text{eval("5")} \longrightarrow \text{Number("5")}$$

$$\text{eval('"S"')} \longrightarrow \text{JSON.parse('"S"')}$$

[3] Although JSON.parse is intended to parse JSON, which is discussed later, string literals are in fact a subset of JSON.

Our approach is to replace *all* numeric and string literals by generalizations which accept all JavaScript numbers and strings, respectively.

Literal objects. As JSON is a safe, pure subset of JavaScript, JSON strings may naturally be passed to eval[4]. The latest version of the ECMAScript standard adds the JSON.parse function, which parses JSON in a pure, safe way, unlike eval. This is the only form of generalization we perform which accepts a non-regular language, though in principle any subset of JavaScript, regular or otherwise, could be generalized.

```
eval('({"S":5})')  ⟶  JSON.parse('({"S":5})')
```

Because JSON is only a form of literals, having no side-effects or external references, this replacement is always safe.

Function arguments. Functions may be called with a variadic number of arguments in JavaScript, and in fact, may be called with a runtime-generated array of arguments of any length with the apply method. If the arguments themselves may be generalized by any of the above methods, then we may further generalize the entire argument list, yielding a recognizer which accepts argument lists of any length.

```
                                        eval('foo(1, 2)')
⟶  foo.apply(window, [Number("1"), Number("2")])
```

4.3 Code generation strategy

Although having a recognizer for those strings passed to a given eval call site is itself useful, the purpose of Evalorizer is to replace eval calls. As such, the next step is to transform the generated recognizers into JavaScript code which will replace eval. In this section we discuss how the replacement code itself is generated. This varies based on the shape of the recognizer tree, so it will be defined incrementally. The general shape of the replacement code for an eval call site is straightforward:

```
if (arg matches pattern) { // Guard test
    result = replacement_code
} else // Fallback case
    result = default_action(arg);
```

The way that the matching and replacement code are generated varies by the nature of the recognizer, so will be discussed in the next sections. The default action is a matter of policy. This action is generally a fallback to eval(arg), which preserves the original semantics of the eval call site, but when this code is produced by the patcher tool (in *strict* mode), it will instead throw an exception (throw evalorizer_exception(arg)). This strict mode allows safe embedding of third party code where the entire allowed subset of JavaScript is known.

[4] For historical reasons, JSON is usually wrapped in parentheses when passed to eval. Removing the parentheses in this case is a trivial optimization.

For simplicity of presentation, the regular expressions shown in this section are simplified to ignore issues of whitespace, and other non-AST-impacting alterations to JavaScript code. The true regular expressions generated by Evalorizer of course are not simplified in this way.

4.4 Constant trees

The most straightforward replacement is for recognizer trees which contain neither choice nodes nor generalizations. In this case, the tree is precisely a JavaScript AST, so Evalorizer can simply deparse it as its own replacement code. All trees without generalizations represent regular languages, so a regular expression can be generated to match the argument; for the case of constant trees, even a regular expression is often more powerful than necessary, a simple string comparison of the argument may be sufficient. In a call site which had only been called with the string "window.width", for example, the generated code is:

```
if (arg === "window.width") {
    result = window.width;
} else
    result = eval(arg);
```

4.5 Choices

In the presence of choice nodes, it is necessary to nest guards and perform different replacements based on the choice. This is not because of the nature of the language matched by the recognizer (which is regular), but because the replacement code must be different depending on which choice is taken. As such, Evalorizer generates a single match case for the entire tree which captures the choice subtree by a grouping within the generated regular expression, then matches the choice subtree in a nested condition. To follow our simple example of eval("window."+x), with only "width" and "height" as values of x and no generalization performed (i.e., Fig. 3c), the generated code is:

```
var re = /^window\.(width|height)$/;
if (match = (re.exec(arg))) {
    if (match[1] === "width") {
        result = window.width;
    } else if (match[1] === "height") {
        result = window.height;
    }
} else
    result = eval(arg);
```

The options may be checked in any order, but since the recognizer was generated from a selection of real eval strings, it is logical to order them by the frequency with which particular subtrees were seen.

4.6 Generalization

Each form of generalization node requires its own strategy for matching and replacement, so each is discussed separately. With the exception of JSON, all generalizations presented

here accept only regular languages. As such, regular expressions are used to match them. The generalization node within the tree is replaced by a safe eval alternative in the replacement code, though the particular alternative is specific to the generalization.

Member expressions. In JavaScript, it is possible to access members of objects by using the map operator. As such, we generalize the right hand side of the dot operator to match JavaScript identifiers[5], and generate code using the map operator. Our simple example, eval("window."+x), with generalization over x, is transformed into the following code:

```
var re = /^window\.([a-zA-Z_$][a-zA-Z\d_$]*)$/;
if(match = re.exec(arg)) {
    result = window[match[1]];
} else
    result = eval(arg);
```

Because the identifier is collected as a string and no choice nodes exist, we need only to generate one replacement, rather than one per all possible identifiers.

Literal primitives. The generalization of literal primitives is straightforward, as the grammars for both numeric and string literals are regular, and safe alternatives to eval for both are known. Consider an example eval("foo("+ x + ","+ y + ")"), where x and y always contain a string and numeric literal, respectively, and so a recognizer which generalizes over numeric and string literals has been generated. The transformed code for this case is[6]:

```
var re = /^foo\(("([^\\"]|\\.)*"),(\d+)\)$/;
if (match = re.exec(arg)) {
    result = foo(JSON.parse(match[1]), Number(match[3]));
} else
    result = eval(arg);
```

Literal objects. JSON.parse parses *only* valid JSON and is hence safe and pure. Our code generator can therefore avoid the complicated parsing of JSON itself, and simply replace eval(arg) with JSON.parse(arg)[7]. The caveat is that JSON is not a regular language, so matching it becomes complicated. Because JSON cannot be matched correctly by a regular expression, we match it in two steps: First we use a very broad regular expression to match a large superset of JSON[8], then we use JSON.parse to validate that the matched substring is in fact JSON. This technique has the implication that a recognizer cannot contain multiple JSON generalizations within the same choice node, as the resultant regular expression would be ambiguous. As an example, consider the common pattern of JSONP, e.g. eval("foo("+ x + ")") where x contains only valid JSON strings. Assuming that the recognizer has formed a generalizer node for x, the code for the transformed eval call site is:

```
var re = /^foo\((.*)\)$/;
if (match = re.exec(arg)) {
    try {
        json = JSON.parse(match[1]);
        result = foo(json);
    } catch (ex) {
        result = eval(arg);
    }
} else
    result = eval(arg);
```

Function arguments. Functions may be called with a runtime-defined variadic number of arguments through the apply method. This generalizer is only used by Evalorizer when the arguments themselves conform to a single generalizer, but in principle could work with any regular arguments. An eval call site which has received the strings "foo(1)" and "foo(2,3)" would generate a recognizer which generalizes the argument list to a variadic list of numeric literals, and transform the eval call site as follows:

```
var re = /^foo\(((\d+,)*\d+)\)$/
if (match = re.exec(arg)) {
    var args1 = match[1].split(",");
    var args2 = [];
    for (var i = 0; i < args1.length; i++)
        args2.push(Number(args1[i]));
    result = foo.apply(window, args2);
} else
    result = eval(arg);
```

4.7 Preserving original program semantics

One of the main concerns of this work is to preserve the original behaviour of websites. The semantics of eval in JavaScript is subject to some oddities. When eval is called directly, the code is evaluated within the enclosing lexical scope. On the other hand when it is called indirectly, i.e., through an alias, the code is evaluated in the global scope. Thus to ensure that our replacement code behaves like the original (variable access and hoisting), the generated code must be inlined and no factorization is possible. Unfortunately, this comes at the expense of the code size. Additionally, because it is not generally possible to know whether any call will indirectly dispatch to eval, and we cannot allow eval itself to be an alias, our tool is currently unable to handle indirect calls to eval.

While eval is capable of evaluating both expressions and statements, it is itself an expression. As such, a difficulty arises when an eval accepts a statement but is embedded into a larger expression. The value returned by eval is the value of the last expression evaluated. Fortunately the comma operator,

[5] As JavaScript allows Unicode characters in identifiers, but JavaScript's regular expression engine is insufficient to specify all valid identifier characters, we only accept ASCII identifiers.

[6] As the regular expressions for string and numeric literals in JavaScript are actually quite complicated, we present a simpler one here.

[7] As JSON strings are usually parenthesized when passed to eval to avoid being parsed as block statements, we strip out these parentheses if present.

[8] Because this regular expression is itself quite complicated, our example shows a univeral match, .*. The ambiguity is still resolved by the presence of try/catch around JSON.parse.

i.e., a sequence separated by comma, has the same behavior and thus can be used as a substitute. Our replacement code make heavy use of this operator. However, only expressions can be used with the comma operator, and eval may contain other forms of statements. In order to handle this case, Evalorizer would need to partially evaluate the context which embeds eval, then inject all of the evaluated statements, and finally finish the outer context. No such case has been encountered on real website, so we currently not support this case, but we do not expect difficulties in doing so should the need arise.

Finally, our replacement code makes use of temporary variables. These variables may collide with user code variables. However, since we can only generate legal JavaScript code, these name clashes are unavoidable. It is arguable that some analysis may check for name conflicts, but due to the dynamic nature of JavaScript (and particularly its global scope), this too could be unsafe. Evalorizer's solution is simply to prefix all its own variables by an unusual pattern, making name clashes highly unlikely in practice.

5. Evalorizer

In this section we describe the architecture of Evalorizer, starting by explaining the proxy part (Sect. 5.1) then the patcher in Sect. 5.2.

5.1 The Proxy

As the name suggests, the Evalorizer proxy is an HTTP proxy. It is implemented in JavaScript and runs on *node.js*[9], a JavaScript runtime based on Google V8[10], designed for writing scalable internet applications such as web servers and web proxies. A proxy allows dynamic instrumentation of JavaScript code that is browser-independent, as well as being server and server-side language independent. Additionally it can work with all means of including JavaScript, i.e., inlined with <script> tags, in event handlers or in external files. Moreover it works with both static and dynamic pages built with any kind of technologies. The proxy does not require any changes to the development, testing or deployment environments since it inspects actual traffic to and from the website, and does not depend on the original eval call string or how it was originally obtained.

Instrumenting a call site. The proxy inspects all the JavaScript code sent to the browser. For every call site, its location is logged. Finding the eval call sites is done by building the AST of all the JavaScript code received from the server. The eval call site node is then instrumented with the logging code (logEval).

Logging invocations. When eval is invoked, the (logEval) code will send an asynchronous HTTP request (XMLHttpRequest) with the argument and the call site identifier. The proxy receives the request and logs it.

5.1.1 Adding short-circuits.

Evalorizer adds to the logEval code a few "short-circuits" for extremely common uses of eval. Since the great majority of eval invocations are used to read variables or deserialize JSON objects, the code for these two cases is also embedded. On a page reload, if the proxy has seen at least one argument for that call site, the proxy also embeds the best recognizer created yet for that call site. The main purpose of "short circuits" is to give an instant intuition to the user if Evalorizer will be able to resolve that call site properly or not and wether sufficient logs have been collected or not. The user can figure out the resolved call sites by inspecting the proxy log. The short-circuits are not needed for the creation of the final patch.

5.2 The Patcher

Starting form the logs collected by the proxy, the patcher applies the techniques discussed in Sect. 4. Since evals are ideally all supposed to be replaced, the fallback may be outright removed. We provide two options for removing the fallback. In a *strict* mode, it is replaced by throwing an exception. This behavior is recommended especially for eval of third party code, as it reduces the possibility of code injection, XSS attacks or other attacks exploiting eval's characteristics. The alternative is to use eval as a *fallback*, it uses eval if it fails to recognize some invoked argument, and this can be used for call sites that were not invoked sufficiently.

The transformation for each page is collected, and applied to the original code. Evalorizer also generates a patch suitable for the Unix `patch` command, in unified format. Finally the programmer may choose if he wants to apply the patch or copy the whole, updated file. Furthermore, as this solution is intended to allow for sound, gradual replacement, the patch can be generated to use a logging facility as its fallback. This requires that the user hosts a light version of Evalorizer on the site's server, but has the advantage of catching missed eval calls with real-world users.

6. Evaluation

We validate Evalorizer with three different experiments. The first one intends to show that our heuristics are legitimate and are more efficient than a brute-force approach to the eval replacement problem. For this a large set of logs collected from the top 100 most popular websites is used as well as some live websites which have significant usage of eval. The second will show that our system performs well even with incomplete sampling of eval strings, i.e., under the open world assumption, by using cross-validation techniques. Finally a third experiment will briefly present some efficiency results while browsing webpages with eval, during instrumentation, and once these pages are patched.

[9] `http://nodejs.org/`

[10] `http://code.google.com/p/v8/`

6.1 Corpus

In most languages, gathering a substantial corpus is difficult because accessibility to source code and runtime configurations is limited by pragmatic and legal issues. On the web, this is generally not the case, as JavaScript is only transmitted in source form. As such, our corpus selection is not a matter of accessibility, but utility: we wish to analyze popular pages which use eval in typical ways.

The largest corpus we used to validate our methodology is the interactive logs collected in our previous work [21]. These logs represent real interactions with the top 100 most popular websites at the time, according to the `alexa.com` list as of March 3, 2011. At least two interactions by two different researchers were performed for each site. Each log represents a human interaction, with each session lasting 1 to 5 minutes and approximating a "typical" interaction with the website, including logging into accounts where necessary. Of these 100 recorded sites, all use JavaScript, and 82 use eval. The logs represent 204MB of unique JavaScript code, a total of 7 078 calls to eval (with an average of 84 eval calls per trace) and a total of 8.2MB of code passed to the eval function (with an average of 1 210 bytes per trace). As this data was collected from the most popular websites, its extensive use of eval is furthermore a validation of the necessity of our technique.

Because our largest corpus is recorded logs, and not live web pages, we could only use it to validate the *technique*, and not the implementation. We selected for further study 4 websites from this list which had considerable use of eval: `cnn.com`, `myspace.com`, `alibaba.com`, `ebay.de`. We cached a few pages from each website, and hosted them locally to test the complete system without upstream changes. These cached portions include 12.2KB, 1.6K, 3.6K and 6.4K of JavaScript code, respectively, and generated 738, 42, 24, and 99 calls to eval. Those numbers are relatively small, but typical, and they represent a complete set for the cases seen from the logs. The cache included more eval call sites, but they were never invoked in this configuration.

In many areas of research relating to JavaScript, two popular suites of benchmarks are used to validate results: SunSpider and V8. Although they can be useful in some circumstances, our focus is on eval, and their use of eval is uninteresting. The former uses eval in four of its included benchmarks, the latter in only one, and neither are representative of real world JavaScript behavior [18, 19].

6.2 Distribution of call sites

The purpose of this first experiment is twofold: we show that the tool works and actually removes eval, and demonstrate that our tree representation and our generalizing strategy are well suited and allow for shorter code than a brute-force approach which would directly inline the code of each eval call string.

Benchmark	Constant		Single-pattern		Multi-pattern	
	cs.	invoc.	cs.	invoc.	cs.	invoc.
CNN	4	37	20	1 364	2	4
MySpace	6	1 135	3	10	1	6
Alibaba	15	578	0	0	1	7
eBay.de	9	26	39	186	1	7
All	179	2 928	411	9 104	112	901

Constant stands for a call site where all its invocations have the exact same string, *Single-pattern* for recognizers which matches a single pattern, i.e., no choice nodes, and *Multi-pattern* for all remaining cases. Numbers for actual websites are reported twice, once in their own row, once in "All".

Table 1. Call sites meta-pattern distribution.

Table 1 reports how many recognizers of each kind have been generated for each call site (static) and how many times they are invoked (dynamic). In this table, "constant" means that the string will be inlined literally and no choice nodes or generalizations exists. Hence those cases are the equivalent of the brute force approach and will be handled by a simple eval cache strategy, which is used in most JavaScript virtual machines. The "single-pattern" category represents recognizers which match a pattern containing some generalization, but no choice nodes; for instance, matching all numbers instead of a single constant. Finally, "multi-pattern" is the most general, covering all remaining cases.

According to these numbers, 25% of the call sites recorded are constants, representing roughly 23% of the invocations. All others recognizers are more general, thus would lead to code explosion if they were handled in this straightforward way. This suggests that programmers are still unaware of alternatives to eval. Multiple-pattern while less useful, is needed for completeness; without it, 7% of the dynamic calls would fall back to eval (and 16% of the call sites). Unsurprisingly, thanks to our approach, once the code is patched, replaying the experiment does not trigger a single eval. Hence the number of mispredictions is always null and thus not included in this table. Moreover we classified all eval call sites of the 4 websites manually and we were unable to find, without changing the surrounding code, a better classification than the one reported by our tool.

Table 2 reports the effect of each generalization. For each generalization, we count each generalization node, and how many time it was invoked. "Other nodes" report the number of non-generalized nodes. Some of the eval call sites were only invoked once despite our best efforts, and we report those separately. We attempted the generalizations on that single argument, since it is very likely that the generalization holds for any other still not encountered argument according to statistics of [19].

All these numbers are significant even when compared to non-generalization nodes, hence having special cases

Category	nodes	invoc.	nodes *appears one*	invoc. *appears one*
Member expressions	2 688	7 862	149	149
Literal primitives	12 904	33 825	5 908	5 908
Literal objects	271	1 260	317	317
Function arguments	105	2 922	113	113
Other nodes	13 181	39 498	9 289	9 289

Table 2. The number of generalization nodes and invocations affected by each generalization.

for these generalizations seems to be a good strategy. The other nodes are used for whole expressions, but most of them are operators standing between generalized categories. From a purely numerical standpoint, the most profitable generalization, far above all others, are literal primitives. This is unsurprising since most eval calls only differ from each other by some constant coming from user input or produced by some code generator. More surprisingly, a *function arguments* generalized node is used on average 14 times, where *member expressions* are are used on average 3 times. However, in absolute terms, *member expressions* are still triggered much more often than *function arguments*. This too is unsurprising since in all languages, and JavaScript is not an exception, reads from members are a very common action. Even if *literal objects* serialization is triggered less than all other generalizations, it is still a good idea to have a special case for several reasons. First, it reduces considerably the size of the recognizer and the code generated; *literal object* strings are generally long and introduce a lot of variability. A second reason, though a less clear one, is from an efficiency standpoint. Since most browsers are faster to parse JSON strings than evaluating them, though Safari is a notable exception, replacing eval at the parsing sites improves the performance of websites that utlize JSON oftenly.

From our perspective, this experiment is a success, since all our generalizations seem adequate as they both reduce the code size and increase the power of generalization of Evalorizer.

6.3 Classification stability

At heart, Evalorizer is a learning algorithm. However, unlike classical learning algorithms, it builds an overfitted classifier to assuredly match all input strings. On our dataset, with only the eval strings we've seen, eval will absolutely never be called, as Evalorizer learns by rote. In order to evaluate Evalorizer's generalization power, it must be evaluated on inputs that were not used in the learning phase. To simulate this, we use the $k-$fold cross-validation technique. For each eval call site, invocation logs are split into k sets of equal size. $k-1$ of these sets are used as input to train Evalorizer. Once trained, the code is patched in strict mode, such that we can determine and catch any failing eval cases. Finally

Method	Mispredict	%	Call sites affected	%
Leave-one-out	215	1.7%	99	18.2%
Holdout	374	2.89%	128	14%

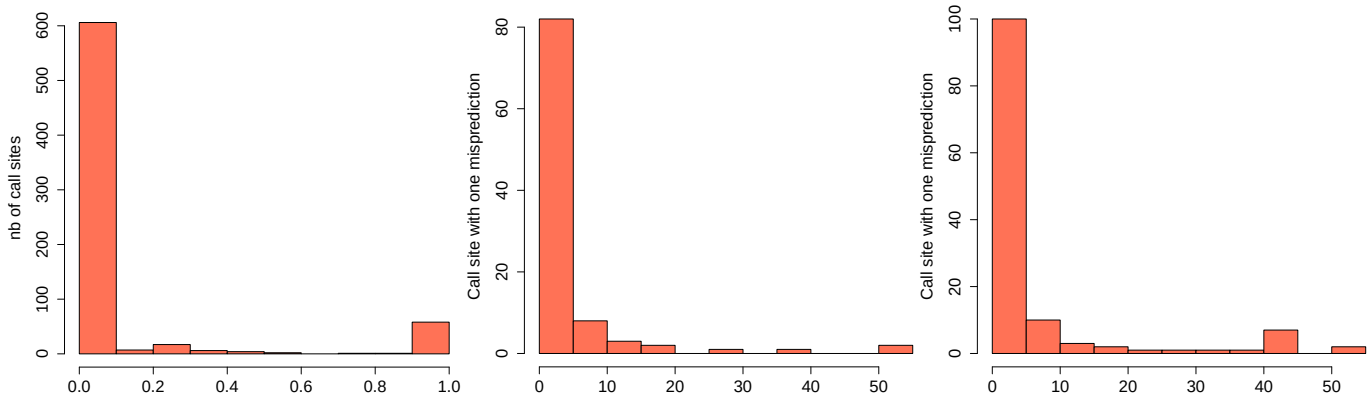
Table 3. Misprediction using two cross-validation methods.

the unused set is used to *validate* the patched code. This experiment is then repeated until each set has been used as a test set. Two special cases of this technique may also be considered, the *Holdout* method, where k is equal to two, and the *Leave-one-out* method, where k is equal to the size of the dataset. We applied both holdout and leave-one-out. The former is not to our advantage since the training set is very small, and the splitting point has a huge impact. The latter may be computationally expensive but in practice takes no more time than computing the residual error and it is a much better way to evaluate models.

Since at least two samples are needed for this experiment, i.e., one as training and one to test, all call sites with only one string available have been removed, and only 702 call sites remain, with 12 933 samples. This represents roughly 37% of the whole dataset. Results are shown in Table 3.

The worst case is with the holdout method. There, 2.89% of input strings are mispredicted. The number is encouraging; clearly most eval replacements are well represented by our system even when the training set is unreasonably small. These mispredictions affected 18.2% of the eval call sites. On the leave-one-out experiment, the results are, unsurprisingly, better; only 1.7% misprediction, distributed on 14% of call sites. The distribution of percentage of errors by call site (Fig. 4a) reveals two extreme cases. Either call sites are perfectly predicted, or Evalorizer fails totally. The straightforward conclusion is that these call sites are highly polymorphic, and not enough samples were available for them. To understand why so many call sites are affected by misprediction, it is necessary to look at the number of samples available on call sites having at least one error. Therefore, Fig. 4b and Fig. 4c shows, respectively for the leave-one-out and the holdout methods, a distribution of the number of samples for a call site which has at least one error. Unsurprisingly, call sites with few samples tend to have more errors in both methods, since if the number of samples is two, these methods are equivalent. We argue that a better acquisition of these call sites, i.e., spending more time collecting eval strings, will solve this problem.

The Fig. 4c has a strange peak at 42. A manual inspection of this particular call site, used on many eBay subdomains, shows that it was used for browser detection. The mispredicted arguments is a call to a helper function which sets up non-standard variables used by some browser. These calls only appear once and very early in a browsing session. However, while the holdout method split the data in two sets, these



(a) % of error (leave-one-out) (b) Number of samples per call sites (leave-one-out) (c) Number of samples per call sites (holdout)

Fig. 4a shows the distribution of misprediction (the x-axis is % of misprediction) by call site for the leave-one-out method. Fig. 4b and 4c show, respectively for the leave-one-out and holdout methods, the number of inputs (on x-axis) for call sites containing *at least* one misprediction.

Figure 4. Distribution of errors.

samples are never seen in both of them. These errors are hence just statistical artifacts, and in a normal Evalorizer session they would be recorded, and would never be mispredicted.

This experiment bolsters our claim that our tool's generalization is robust enough to work on true websites. Therefore we consider that our tool is sufficient to remove eval on real code and may be, even in its current prototype version, used in a production environment.

6.4 Runtime efficiency

Although the aim of Evalorizer is to advise web developers in the replacement of eval, it is arguable that, to be useful, the runtime overhead of our tool needs to be low. Therefore, we measured the speed of pages while using our tool. However, since most websites do not use eval frequently, and as such would not have their performance affected at all, we only concentrate on the web page which our previous study showed the most usage of multiple kinds of eval Table 1, namely CNN. We claim that if this web page, as the worst case, is not affected, other web page will not be either. To measure runtime of this site, we used JSBench [20] to generate a deterministic benchmark from CNN website, and use it to measured run times. This experiment was run on a 2.4 GHz Intel Core i5 Mac with 4GB 1067 MHz DDR3 RAM running Mac OS X Lion. Three browser were considered: Google Chrome 18.0.973.0, Safari 5.1.12 and Opera 11.60.

Table 4 shows that while eval code is instrumented, the performance is degraded, but by a small amount. Most of this overhead is actually due to the systematic read and JSON patterns short-circuits. A simple solution to most of this overhead could simply consist of disabling these short-circuits in the Evalorizer proxy. Since eval hampers most static analysis made by JavaScript virtual machines, it makes sense to see if once the code is patched some runtime efficiency improvement may be expected on the website.

	Original		Instrumented			Patched		
Browser	avg.	σ	avg.	σ	ovh.	avg.	σ	ovh.
Chrome	139	50	224	72	61%	115	27	-18%
Safari	166	8	231	19	39%	153	10	-8%
Opera	265	17	345	39	30%	285	13	7%

Average time (avg.) and standard deviation (σ) are in milliseconds. Overhead (ovh.) is expressed as a percentage.

Table 4. Runtime efficiency and overhead of Evalorizer on CNN website.

This is actually confirmed on most browsers, opera being an exception, however the difference is close to the confidence interval. As this is a prototype, we did not focus on optimizing our code generator. Even as such, once patched, only 369 bytes of eval replacements (less than 0.015% of the total JS code) is added to CNN. We expect this overhead to be acceptable for all practical purposes.

7. Related Work

The implementation of Evalorizer is based on the JSBench framework [20]. This work was motivated by the results of and evaluated by the framework of our previous analysis of how eval is used in JavaScript [21] [19]. The combination of simple patterns and consistency are what motivated the present paper.

Work to guide static typing by dynamic instrumentation of eval has been done in the context of Ruby by Furr et al. [8], but this work makes no attempt to generalize beyond seen eval strings, as in that context the range of strings seen at any call site is very small. Furthermore that work does not to our knowledge make any attempt to merge seen strings, resulting in verbose replacement code. Jensen et al. [14] use a technique similar to ours in static analysis of Java-

Script, but their technique does not include a mechanism similar to generalizations, and so with the exception of certain specializations, they are only able to accept particular strings which their static analysis framework indicates are possible. Since their technique uses static analysis, this list is sound and complete when attainable, but cannot apply to some particularly treacherous cases of eval. We know of no reason why our techniques cannot complement each other to create a more broadly applicable and robust hybrid analysis approach.

Some papers apply strength reduction of eval in order to better perform analyses. The analysis done for AJAX intrusion detection of Guha et al.[10] converts unanalyzable eval calls into analyzable JSON parsing by intercepting client requests. But their solution only looks at JSON. Our analysis complements this by converting more general classes of eval calls into statically analyzable equivalents. They use a similar technique to intercept responses, but do not provide replacements, instead their proxy has to be enabled all the time to prevent the intrusion. A different approach to prevent possible malicious behavior of eval is the sandboxing of unsafe code including eval from untrusted sources. The solution by Dewald el al.[5] proposes a framework for sandboxing unsafe code, but unlike Evalorizer it requires changes to the browser. The code is not changed by this approach, and therefore does not enhance results of static analysis on such code. The runtime will also run slower due to the sandboxing overhead.

Many other works have simply ignored or outlawed eval altogether. Gatekeeper's [9] enforcement of security policies depends on the outright exclusion of eval, as does Maffeis et al.'s isolation technique [16]. The latter's framework provides the ability to define wrappers for the eval that parses and rewrites the argument to add the necessary checks for safety to create a safer argument for eval, but making that general would require running a full JavaScript parser written in JavaScript on the client machine each time eval is invoked, therefore adding a large overhead. Unlike the proposal in [16], the final patched version from Evalorizer has no overhead, if a slight speed-up, since the analysis itself was done earlier on the proxy during the acquisition phase. Moreover the both of those propositions have a practical limitation: untrusted code must not use eval. This limitation is severely limiting by design, since untrusted code has to be considered always unsafe and may do whatever it wants.

There are methodologies such as Chugh et al. [4] which take into consideration eval while doing analysis. This staged information flow technique starts by doing a classic static analysis without considering eval, then when eval is actually invoked, its value is propagated and a partial or whole analysis is done again. This hybrid methodology has to punt static analysis to a dynamic one, where the replacement of eval in our approach yields normal JavaScript code with few or no eval calls for future static analysis, considerably different goals.

Another more subtle concern with eval in research is that although several formal semantics for JavaScript exist [15] [11], none to our knowledge include support for eval. Although not a significant issue with the semantics themselves, this has the side effect that any further research using these formal semantics as a base is unlikely to have expected results when eval is present. Since our solution is to remove eval from code, the results produced by Evalorizer are compatible with these formal semantics.

8. Conclusions

Despite the fact that eval is dangerous and mostly useless, many people still use it. This is likely because of its ease of use, legacy code which is difficult to change, or because they are not sure what will be evaluated, for instance because it came from third party code. Therefore we proposed in this paper Evalorizer, a tool and technique which dynamically inspects arguments of eval and suggests replacements. This tool does not require any knowledge of the actual code, nor changes to the browser or to the server. This tool is based on a simple key idea, that a call site generally has only one purpose, which is confirmed by the observation that most eval call sites receive arguments which are the same or very similar. Evalorizer customizes each eval call site according to actual values passed to eval, while trying to generalize it as much as possible with JavaScript's features. Evalorizer has been evaluated both on extensive logs from the 100 most used websites and on four sites which have significant use of eval. This evaluation showed that more than 97% of eval invocations have been replaced by our tool under open world assumption, and this even with a small training set. Moreover the slowdown using this tool is reasonable, and once eval has been definitely replaced, slight performance gain may be expected. According to these experiments, we conclude that this simple idea works surprisingly very well on true websites and actually and practically removes calls to eval. We see another application to this work to JavaScript virtual machines. Many virtual machines cache eval strings for performance reasons. Replacing this simple cache by our technique may improve eval efficiency.

Acknowledgments. We thank Ben Livshits, Anders Møller and Peter Thiemann for fruitful discussions, and the anonymous reviewers for their comments. This work was partially supported by a SEIF grant from Microsoft Research, a Fellowship from the Mozilla Corporation and the NSF grant OCI-1047962.

References

[1] Christopher Anderson and Sophia Drossopoulou. BabyJ: From object based to class based programming via types. *Electr. Notes in Theor. Comput. Sci.*, 82(7):53–81, 2003. doi: 10.1016/S1571-0661(04)80802-8.

[2] Christopher Anderson and Paola Giannini. Type checking for JavaScript. *Electr. Notes Theor. Comput. Sci.*, 138(2):37–58, 2005. doi: 10.1016/j.entcs.2005.09.010.

[3] Michael Bolin. *Closure: The Definitive Guide*. O'Reilly Series. O'Reilly Media, 2010. ISBN 9781449381875. URL http://books.google.ch/books?id=p7uyWPcVGZsC.

[4] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *Conference on Programming language design and implementation (PLDI)*, pages 50–62, 2009. doi: 10.1145/1542476.1542483.

[5] Andreas Dewald, Thorsten Holz, and Felix C. Freiling. ADSandbox: sandboxing JavaScript to fight malicious websites. In *Proceedings of the Symposium on Applied Computing (SAC)*, 2010. doi: 10.1145/1774088.1774482.

[6] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2009. doi: 10.1007/978-3-642-02918-9_6.

[7] European Association for Standardizing Information and Communication Systems (ECMA). *ECMA-262: ECMAScript Language Specification*. Fifth edition, December 2009. URL http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[8] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2009. doi: 10.1145/1640089.1640110.

[9] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, 2009. URL https://www.usenix.org/events/sec09/tech/full_papers/sec09_javascript.pdf.

[10] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for Ajax intrusion detection. In *Conference on World wide web (WWW)*, 2009. doi: 10.1145/1526709.1526785.

[11] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010. doi: 10.1007/978-3-642-14107-2_7.

[12] Dongseok Jang and Kwang-Moo Choe. Points-to analysis for JavaScript. In *Proceedings of the Symposium on Applied Computing (SAC)*, 2009. doi: 10.1145/1529282.1529711.

[13] Simon Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Symposium on Static Analysis (SAS)*, 2009. doi: 10.1007/978-3-642-03237-0_17.

[14] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2012. doi: 10.1145/2338965.2336758.

[15] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *Symposium on Programming Languages and Systems (APLAS)*, 2008. doi: 10.1007/978-3-540-89330-1_22.

[16] Sergio Maffeis, John Mitchell, and Ankur Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *Computer Security – ESORICS 2009*, 2009. doi: 10.1007/978-3-642-04444-1_31.

[17] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the design of the R language. In *European Conference on Object-Oriented Programming (ECOOP)*, 2012. doi: 10.1007/978-3-642-31057-7_6.

[18] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. In *Conference on Web Application Development (WebApps)*, 2010. URL http://www.usenix.org/events/webapps10/tech/full_papers/Ratanaworabhan.pdf.

[19] Gregor Richards, Sylvain Lesbrene, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the ACM Programming Language Design and Implementation Conference (PLDI)*, 2010. doi: 10.1145/1806596.1806598.

[20] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of JavaScript benchmarks. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2011. doi: 10.1145/2048066.2048119.

[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 *European Conference on Object-Oriented Programming (ECOOP)*, 2011. doi: 10.1007/978-3-642-22655-7_4.

[22] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Annual Computer Security Applications Conference (ACSAC)*, 2010. doi: 10.1145/1920261.1920267.

[23] Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *European Symposium on Programming (ESOP)*, 2005. doi: 10.1007/978-3-540-31987-0_28.