The Eval that Men Do

A Large-scale Study of the Use of Eval in JavaScript Applications

Gregor Richards Christian Hammer Brian Burg[†] Jan Vitek

Purdue University [†] University of Washington

Abstract. Transforming text into executable code with a function such as JavaScript's eval endows programmers with the ability to extend applications, at any time, and in almost any way they choose. But this expressive power comes at a price. Reasoning about the dynamic behavior of programs that use this features becomes difficult. Any ahead-of-time analysis, to remain sound, is forced to make pessimistic assumptions about the impact of dynamically created code. This pessimism affects the optimizations that can be applied to programs, significantly limits the kinds of errors that can be caught statically and the security guarantees that can be enforced. A better understanding of how eval is used could lead to increased performance and security. This paper reports on a large-scale study of the use of eval in JavaScript-based web applications. We have recorded the behavior 317 MB of strings given as arguments to 481,844 calls to the eval function. We provide statistics on the nature and content of strings used in eval expressions, as well as their provenance and data obtained by observing their dynamic behavior.

eval is evil. Avoid it.
eval has aliases. Don't use them.
—Douglas Crockford

1 Introduction

JavaScript, like many dynamic languages before it, makes it strikingly easy to turn text into executable code at runtime. The language provides the eval function for this purpose (as well as a few other entry points such as setInterval, setTimeout and Function). While eval is undeniably a strength of JavaScript, as attested to by its widespread use, it is also a thorn in the side of anyone intent on providing static guarantees about the behavior of JavaScript code. It may be argued that correctness and efficiency are not primary concerns of designers of web applications, but security has proven to be harder to ignore. And, as web applications become central in our daily computing experience, correctness and performance are likely to gain in importance.

See no Eval, Hear no Eval. The actual use of eval is shrouded in myths and confusion. A common Internet meme is that "eval is evil" and thus should be avoided.[1] This comes with the frequent assertion that eval is the most misused feature of the language.[2] Although eval is a significant feature of JavaScript, it is common for research on JavaScript to simply ignore it [2,12,24,1], claim it is hardly used (only in 6% of 8,000

[1] `http://javascript.crockford.com/code.html`
[2] `http://blogs.msdn.com/b/ericlippert/archive/2003/11/01/53329.aspx`

programs according to [8], but 44% of websites in [19]) or assume that usage is limited to a relatively innocuous subset of the language [9,14]. In [9], eval is cited as being used primarily for JSON deserialization and sometimes loading of library code, but unspecified dynamic methods are required for evals which cannot be statically determined to be in one of these categories. [14] takes this assumption further by ignoring the effects of eval and producing a warning. The security literature views eval as a serious threat [23]. Although some systems have unique provisions for eval and integrate it into their analysis [7], most either forbid it completely [17], assume that its inputs must be filtered [5] or wrapped [13], or pair a dynamic analysis of eval with an otherwise static analysis [4].

True Eval. The goal of this paper is to conduct a thorough evaluation of the real-world use of eval. We want to settle the debate as to the frequency of dynamic and static occurrences of eval in web applications. To this end, we have built an infrastructure that automatically loads over 10,000 web pages. Though simply loading a web page may trigger the execution of non-trivial amounts of JavaScript, such *non-interactive* executions are not representative of typical user interactions with web pages. In addition to page-load program executions, we use a random testing approach to automatically generate events that explore the state space of web applications. Lastly, we have also interacted manually with approximately 100 web sites. Manual interaction is necessary to generate meaningful interactions with the websites. For all web page executions, we have obtained behavioral data with the aid of an instrumented JavaScript interpreter. We pay particular attention to program source, string inputs to eval, *provenance* information for those strings, and the operations performed by the eval'd code (such the scopes of variable reads and writes).

While this paper focuses on JavaScript, eval is hardly unique to that language. Java supports reflection with the java.lang.Reflect package and the class loading infrastructure allows programs to generate and load bytecode at runtime. Dynamic languages such as Lisp, Python, Ruby, Lua, and others invariably have facilities to turn text into executable code at runtime. In all cases, the use of reflective features is a challenge to static analysis. JavaScript may represent the worst case as there is very little that dynamically created code cannot do. Our results reveal the current practice and use of reflective features in one of the most widely-used dynamic programming languages. We hope our results will serve as useful feedback for language implementers and designers.

The contributions of this paper are as follows:

- Extension of the tracing infrastructure described in [22] to support tracing provenance of string data and monitoring scope depth of variable accesses. Addition of tools for automatically loading web sites and generating events.
- Execution traces of a corpus of over 10,000 websites.
- A database summarizing behavioral information gleaned from the above traces, including all input string arguments to eval, string provenance information, and other execution statistics.
- A detailed analysis of the data providing the most thorough study of the usage of eval in real-world programs to date.

Our tools and data are freely available at the project web page:

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

2 The Nature of Eval

The language standardized as ECMAScript [6] has various dialects referred to by names such as JScript, ActionScript, and JavaScript. In this paper, we focus on JavaScript, which is a variation and extension to ECMAScript originating from Mozilla, now implemented in some form in all major web browsers. It was designed in 1995 by Brendan Eich at Netscape to allow non-programmers to extend web sites with client-side executable code. JavaScript can be best described as an imperative, object-oriented language with Java-like syntax, and a prototype-based object system. A JavaScript object is a set of properties, a mutable map from strings to values. A property that evaluates to a closure and is called using the context of its parent object plays the role of a method in Java. Each object has a hidden prototype field[3] which refers to another object. The JavaScript object system is extremely flexible, so it is difficult to constrain the behavior of any given object. For example, it is possible to modify the contents of any prototype at any time or to replace a prototype field altogether. In JavaScript, any function can be a constructor for a "class" of objects, and contains a prototype field, initially referencing an empty object. The new keyword creates an object based on the prototype field and using the function as a constructor. The semantics of new is simple, but unusual: first, an empty object is created, with its parent set to the object referenced by the prototype field of the constructor function. The constructor is then called with this bound to the new object. The return value from the constructor is discarded, and the new expression evaluates to the new object which was bound to this in the constructor.

One of the most dynamic features of JavaScript is the eval construct, which parses a string argument as JavaScript code and immediately executes it in either the local or global scope. There are other similar means of turning text into code in JavaScript, including the Function constructor, functions such as setInterval, setTimeout, and indirect means of adding <script> nodes to the DOM such as document.write and document.createElement. This paper focuses on eval as a representative of this class of techniques for dynamically loading program source at runtime.

The Root of All Evals. Eval can be traced back to the first days of Lisp [20] where eval provided the first implementation of the language that, until then, was translated by hand to machine code. Since then, it has been included in many different forms in many programming languages. Regardless of the language, eval excels at enabling interactive development environments, and makes it easy to extend programs at runtime with little effort.

The Power of Eval. In the JavaScript programming language eval is a function defined in the global object. It takes a single argument, a string. When invoked, it executes the argument and returns the result of the last evaluated expression, or, if an exception was thrown, propagates it. eval can be executed in two ways: If it is called directly, i.e. using the property of the global object in which it is declared, the executed code has access to all variables in scope (local and enclosing) at the point of the call. For an indirect

[3] Though nonstandard, many JavaScript implementations expose this field under the name __proto__. This field is distinct from the *prototype* field of constructors.

call, where eval is stored into a variable which is subsequently called, it executes in the global scope [6, sect. 10.4.2]. The JavaScript language does not offer much in terms of encapsulation, isolation or access control. Thus, code that is run within an eval has the ability to reach widely within the state of a running program and make arbitrary changes. As such, an eval can install new libraries, add or remove fields and methods from existing objects, change the prototype hierarchy, etc.

To illustrate the power of eval, consider the following example, which is one way to implement objects using only functions and local variables.

```
Point = function() {
  var x=0; var y=0;
  return function(op,sel,val) {
    if(op=="r") return eval(sel);
    if(op=="w") return eval(sel+"="+val);
  }
}
```

Every invocation of the function Point returns a new closure which has its own local variables, x and y, that play the role of fields. Calling the closure with "r" causes the eval to read the 'field' name passed as second argument while a "w" causes a side effect.

```
p = Point();
p("r","x"); // returns 0
p("w","y",3); // set y to 3
p("r","y"); // returns 3
```

Thus calling eval exposes the local scope to any string passed to it, potentially breaking modularity if scopes are used to provide encapsulation. Exposing the local scope can be avoided by aliasing eval, as code passed to an aliased eval will run in the global scope. However, in either case the global scope is unavoidably exposed. Any assignment to an undeclared variable in JavaScript, including in an eval, will implicitly declare the variable in the global scope, becoming globally visible and polluting the namespace. For instance, executing eval("x=4") in a scope that does not include x will declare x in the global scope and assign it the value 4.

Necessary Eval? Eval is hardly the only dynamic feature of JavaScript. Even without it the language is highly dynamic, as it places no constraints on modification performed to objects, it has no typing, many operations that would throw exceptions in e.g. Java simply return the value undefined, and strings can be used to index objects. This last property gives much of the power of the reflective features found in languages such as Java. Given all of these other features, the use of eval is often unnecessary. The following is a typical example of misuse:

```
eval("Resources.message_" + validate(userInput))
```

The programmer presumably has some object stored in Resources holding a number of messages. To select the right message at runtime, a string such as "Resources.message_stdError" is assembled out of a constant string and some user input. To be on the safe side, a method is called to try to validate userInput to prevent a code injection attack. Validating user input is tricky and a large number of code injection attacks

come from faulty validators. Of course the above code could be implemented straightforwardly without eval as:

```
Resources["message_" + userInput]
```

Rather than invoking the full power of eval, the above code uses a constructed string to index Resources. This achieves the same effect with none of the security risks. In a language like Java, this problem would be solved by a HashMap.

Another use of eval is tied to data exchange. Unlike Java, JavaScript does not have its own built-in serialization/deserialization facility[4]. However, eval can be used for this purpose. JSON (JavaScript Object Notation)[5] is syntax designed to provide a portable way for applications to serialize and deserialize data. JSON is also, by no coincidence, a subset of JavaScript's object, array, string and number literal syntax, and as such can be deserialized by eval. An example JSON string is:

```
{ "Image": { "Title": "View from 15th Floor", "IDs": [116, 943, 234, 38793],
    "Thumbnail": { "Height": 125, "Width": "100" }}}
```

Pure JSON is fairly restrictive; e.g. {"foo":0} is valid, but {'foo':0} or {foo:0} are not, though all are valid JavaScript and when eval'd have the same semantics. Anecdotal evidence suggests that pure JSON and almost-JSON are both commonly used by developers. JSON is also commonly eval'd along with an assignment of the JSON string to a variable, e.g. x = {"foo":0}. Performing the assignment within the eval is always unnecessary because eval returns the result of the evaluated expression. The canonical scheme to parse JSON and assign it to a variable would be x = eval(y), which assigns the anonymous object created by evaling the JSON in y to x. Modern browsers provide the JSON.parse API as an alternative to eval for pure JSON as well as JSON.stringify for serialization.

Scripts, Eval and Synchronicity. Libraries loaded conventionally by including <script> tags in the document have the unfortunate property of blocking the rendering of the page while the script is downloaded, parsed, and evaluated. This blocking is necessary to ensure deterministic page loading since scripts could arbitrarily modify the DOM before other DOM elements or scripts can be loaded. Although Internet Explorer supports the "defer" property of script tags to avoid this blocking, and HTML5 introduces new script properties for this purpose as well, in the general case it cannot be avoided by HTML alone. As such, a common alternative is to download the script asynchronously with XMLHttpRequest (AJAX), then load it with eval at some arbitrary time. This mechanism does not block page parsing or rendering, but leaves to programmer the burden of ensuring a known, consistent execution state. Code loaded later must also be careful in its assumptions about execution state and other code loaded.

[4] ECMAScript's 5th edition introduces the JSON object for this purpose, but since it is not yet available on all browsers, portable code cannot use it.

[5] http://www.ietf.org/rfc/rfc4627

3 Methodology

In this section we describe the infrastructure used in our experiments and the methodology used to collect our data.

3.1 Infrastructure

The data presented in this paper was recorded using TracingSafari, an instrumented version of the open-source layout engine WebKit[6] running in the Safari 5 browser based on [22] that is able to record low-level, compact JavaScript execution traces. We extended the tracing infrastructure for recording properties particular to eval. In particular, we now support tracking provenance of strings, as these might eventually become arguments to eval. We distinguish the equivalence classes AJAX, native, constant, and combinations thereof. For flexibility, trace analysis is performed offline. TracingSafari records a trace containing most operations performed by the interpreter (reads, writes, deletes, calls, defines, etc.) as well as events for garbage collection and source file loads. Invocations to eval trigger an event similar to the one for source file loads, and the evaluated string is saved and its execution traced like any other part of the program's execution. Complete traces are compressed and stored to disk. Traces are analyzed offline and the results are stored in a database which is then mined for data. The offline trace analysis component performs relatively simple data aggregation over the event stream. For more complex data, it is able to replay any trace creating an abstract representation of the heap state of the corresponding JavaScript program. The trace analyzer maintains rich and customizable historical information about the program's behavior, such as access history of each object, call sites and allocation sites, and so on.

3.2 Corpus

Gathering a large corpus of programs is often difficult because accessibility is limited. With JavaScript this is not the case as almost any non-trivial web site on the Internet uses JavaScript, and all JavaScript is transmitted in source form. In most cases, non-trivial amounts of JavaScript are run automatically as the result of loading a web page in the browser. After that, further program execution is event-driven: event handlers are triggered by user input events such as mouse movements, clicks, and the like. To capture a wide range of behavior we have compiled a corpus based on recording three kinds of executions:

INTERACTIVE	Manual interaction with web sites.
PAGELOAD	Data set obtained by recording JavaScript behavior for 30 seconds when a web page is loaded.
RANDOM	Data set obtained by recording 30 seconds of page load activity and randomly generated events.

All of our runs were based on the most popular web sites according to the `alexa.com` list as of November 29, 2010. INTERACTIVE was generated by hand by recording visits

[6] `http://webkit.org` Rev. 73337.

to the 100 most popular web sites on the Alexa list. The runs were 1 to 5 minutes long and involved what we intend to be typical interaction with the web site, including logging into accounts. PAGELOAD and RANDOM were based on the 10,000 most popular web sites on the Alexa list. PAGELOAD is intended to record the load-time behavior of pages. It simply navigates the browser to each page and records execution for a total of 30 seconds without any further interaction; since scripts are often ongoing, there is no clear indication that a page load has terminated. In this case, a simple timeout is the most reliable way to include load-time behavior without interaction. RANDOM behaves similarly to PAGELOAD, but includes a script which will randomly trigger click events on DOM elements with mouse event listeners registered and click links. The final data was recorded between December 1st and December 10th, 2010. All recorded traces are available from our project's site.

The rationale for these three data sets is as follows: INTERACTIVE provides the best picture of complete interactions with a web application and is thus the most representative of the usage of eval in JavaScript programs. PAGELOAD and RANDOM give us breadth of coverage and allow us to study a much larger number of web sites but with a caveat of reduced program behavior coverage. PAGELOAD will not generate unrealistic behavior, although it may generate atypical behavior. RANDOM, being strictly random, can generate unrealistic behavior, but is the best means of generating a wide variety of behaviors on a large corpus of sites.

3.3 Threats to validity

JavaScript and the DOM provide other means to inject code at runtime, such as the document.write of a script tag, or document.createElement("script"), but these methods are entirely reliant on the browser. Since our tracing infrastructure instrumented only the JavaScript interpreter, we do not have data on their use. Secondly, as with any tracing-based methodology it is difficult to obtain exhaustive coverage. The problem is compounded by the interactive nature of web applications which are driven by the user interface. Furthermore, as programs are only fed to the browser one page at a time, it is difficult to even assess which fraction of a web site was exercised. Our results may thus fail to uncover some interesting behaviors. This said, we believe that the data reported in this paper is representative of typical browsing behavior. A third threat comes from our focus on client-side web applications. It is likely that other categories of applications would display different characteristics. For instance, the widgets of [8] appear to do so. But the importance of web applications and the quantity of JavaScript code on the web mean that this is a class of applications worth studying. Lastly, neither WebKit nor Safari hides its identity to JavaScript code, and as such it is possible for code to exhibit behavior peculiar to WebKit or Safari. Although this does introduce a possible bias which is difficult to detect, all other JavaScript implementations are equally detectable and introduce comparable bias.

4 Usage Metrics

This section presents a high-level picture of the usage of JavaScript and eval in a broad selection of web pages, which is summarized in Table 1. The prevalence of JavaScript

Table 1. Usage statistics.

Data Set	JavaScript used	eval use	Avg eval (bytes)	Avg eval calls	total eval calls	total eval size (bytes)	total JS size (MB)
INTERACTIVE	100%	59%	1486	38	2,434	3,616,822	59.8
PAGELOAD	91%	41%	685	28	111,866	76,669,599	1,725
RANDOM	91%	43%	687	85	367,544	252,340,684	1,829

on the web has increased even further since our previous study [22]. At that time, 97 out of the top 100 sites used JavaScript whereas this time all of these pages use some JavaScript. For the 10,000 most accessed web sites we found that a staggering 91% rely on JavaScript. As for eval, it used widely and frequently in our corpus. We have recorded 481,844 calls to eval for a total of 317 MB of string data. Over 59% of the top 100 pages use it, and 43% of the other pages do as well. It is noteworthy that the difference in the use of eval between RANDOM and PAGELOAD is only 2% which suggests that sites that rely on eval do it even for non-interactive reasons.

Fig. 1 shows the distribution of eval call sites per website, for those sites that call eval at least once. It is unsurprising that the mean number is rather low and that the INTERACTIVE and RANDOM run have higher means as those load several pages per site, while PAGELOAD only loads one page per site. However, it is notable that even for a single page per site, the maximum number is 80, so there is a significant spread. Given that we do not report on other variants of eval, such as the Function constructor, these numbers are to be take as a lower bound on the amount of dynamically created code.

Fig. 2 displays the distribution of the total JavaScript code size loaded for each data set. As in our previous study, it shows that while most web site have less than 512KB of JavaScript code, there are some significant outliers, especially in the most popular sites.

Fig. 3 shows the distribution of string sizes passed as parameters to the eval function. The graph shows cumulative proportions of string sizes. While for INTERACTIVE about two thirds of the strings are less than 64 bytes long, the maximum observed size was 193 KB. The PAGELOAD and RANDOM data sets tell similar stories, 80% and 85%, respectively, of strings are less than 64 bytes, but both peak at 413 KB. This suggests that the data fed to eval can be rather large. The average eval size is 1,486 bytes for the INTERACTIVE, 685 bytes for the PAGELOAD, and 687 bytes for the RANDOM runs, which suggests that the amount of dynamically created scripts is higher in the most popular pages.

Fig. 4 shows the distribution of events intercepted during eval execution for our three data sets. The mean number is again fairly low around 5, with the third quartile going up to about 10 events. The spread, however, is extreme with the RANDOM runs intercepting around 1.4 mio events. Given the maximum size of the strings passed to eval reported in Fig. 3 this size is not all too surprising, though. The maximum number for the INTERACTIVE runs, in contrast, is low compared to its maximal size of eval strings.

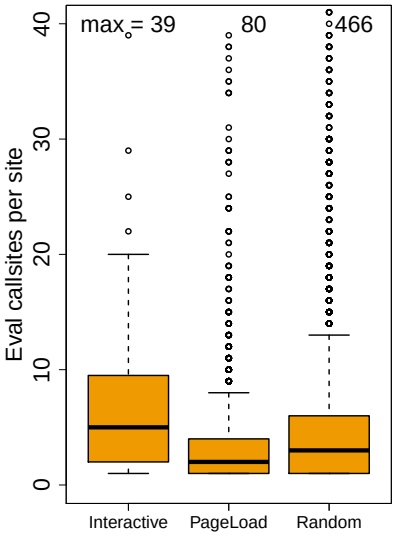


Fig. 1. Distribution of number of eval call sites per site, for those sites that call eval at least once.

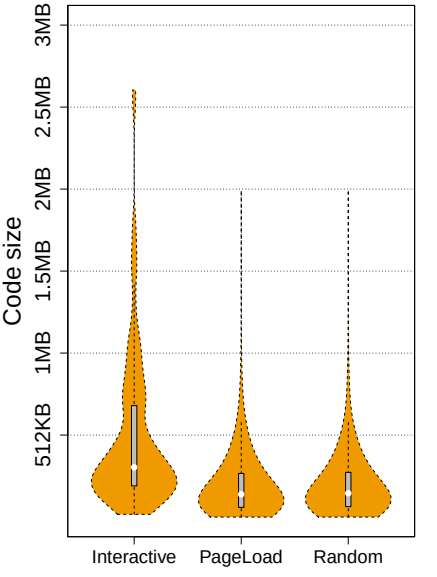


Fig. 2. Distribution of total size of JavaScript code per data set.

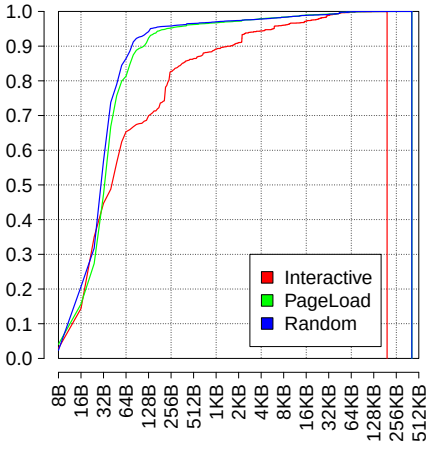


Fig. 3. Distribution of eval string sizes. The y-axis is the cumulative proportion, and x-axis is the size, log-scale. Drop offs indicate the maximum size for each data set.

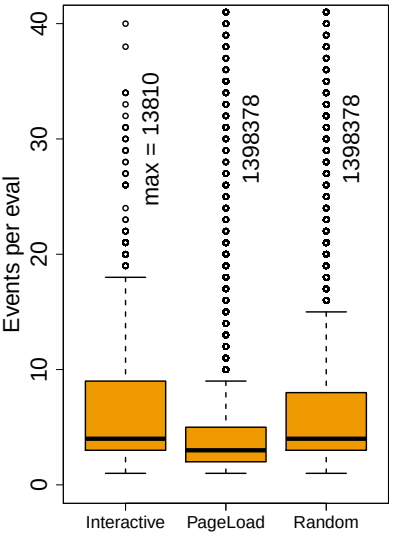


Fig. 4. Distribution of events intercepted in eval code.

In terms of the number of eval we found in those runs, on average we had 38 evals in the INTERACTIVE data set, 28 evals at PAGELOAD, and 85 evals during the RANDOM runs. Clearly, web sites do call eval during the whole life cycle of a web page, not only during page load, nor merely during page interaction. The average of the INTERACTIVE being in the middle of PAGELOAD and RANDOM is explained by the fact that RANDOM is being able to fire more events in a short period of time than a user interaction with the real site would.

Fig. 5 gives the distribution of operation types seen dynamically during eval interpretation for each of our three data sets. We report on stores (STORE), reads (READ) and deletes (DELETE) of object properties (includes indexed (x[3]) and hashmap style (x["foo"]) access), function definitions (DEFINE), object creations (CREATE), and function calls (CALL). Compared to the distribution of operation types we presented for entire JavaScript runs on top 100 sites in [22], the PAGELOAD data set is quite consistent. This suggests that the code executed in eval is not dissimilar to general JavaScript code. In particular, eval is not just parsing of JSON objects, as some related work assumes. It is interesting to note that INTERACTIVE runs contain considerably more store and create events, which we attribute to JSON-like constructs. We will consider the proportion of JSON-like constructs in more detail in Sect. 5.2. The RANDOM runs, on the other hand do more function invocation as part of handling the randomly generated mouse events and again contain some amount of JSON.

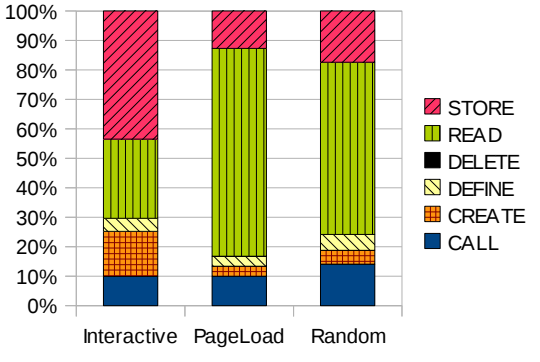


Fig. 5. Mix of operation types seen dynamically during the execution of eval calls.

Table 2 gives the proportion of the sites using common libraries. We found three libraries were used often, namely jQuery, Prototype, and MooTools.[7] By far the most widespread is jQuery, which is present in more than 50% of all websites that use JavaScript. It is, however, interesting to note that some libraries are only loaded when needed for the dynamism of the page. This is shown by the spread between the PAGELOAD and RANDOM runs (for instance 48% and 52% for JQuery). Other libraries commonly found are below 10%, MooTools seems to be more popular in the top 100 sites than in the rest

[7] `jquery.com`, `prototypejs.org`, `mootools.net`, `code.google.com/closure`

Table 2. Common libraries used in our executions.

Data Set	jQuery	Prototype	MooTools
INTERACTIVE	54%	9%	10%
PAGELOAD	48%	6%	4%
RANDOM	52%	7%	5%

of the top 10k. The Google Closure library, which is used by many sites owned by Google, is not automatically detectable, as the Closure compiler does whole-program analysis and obfuscation, so there was not reliably a common string we could use to classify source as using it.

5 A Taxonomy of Eval

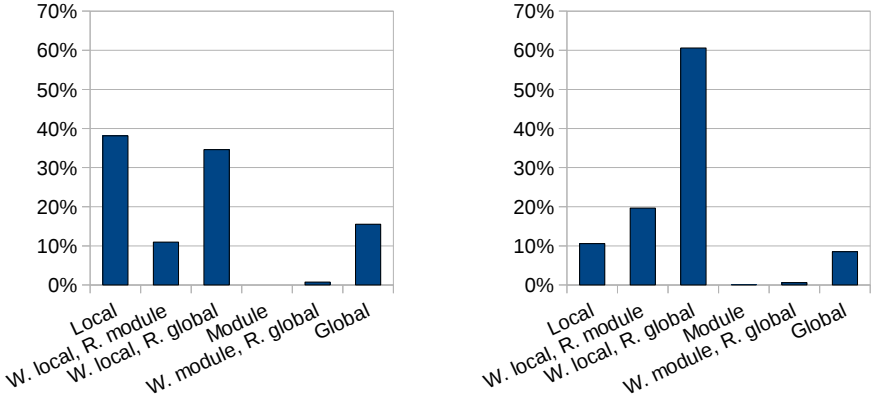
The previous section gave a high-level view of the frequency of eval; we now focus on categorizing the behavior of eval and its impact on program analysis, verification and optimization techniques. There are four important axes to consider. Firstly, we look at what **scope** is affected by operations inside eval'd code. Operations that mutate shared data are more likely to invalidate assumptions or pose security risks than operations that are limited in scope to data created within the eval. Secondly, we try to identify **patterns** of usage. A better classification of the patterns of eval usage can help language designers provide limited, purpose-specific alternatives to eval, and also provide a better understanding of the range of tasks done within evals. Thirdly, we investigate the **provenance** of the string passed into eval. This comes directly from a desire to better evaluate the problems linked to code injection attacks. Our last axis is **consistence**, or how the arguments to a particular eval call site vary from invocation to invocation. We focus on each axis independently, discussing the relationships between them when relevant, then discuss the implications of each on analyses and other systems.

5.1 Scope

As with any JavaScript code, the code executed via eval may access both local and global variables. Code that does not access global state is often preferred. We categorize the locality of accesses within each call to eval into the following sets.

- **Purely local.** The eval accesses nothing, or only accesses variables in the enclosing scope. For example, JSON only creates new objects and modifies these in the course of its parsing.
- **Writes local, reads module.** The eval writes only to local variables, but reads from outer scopes, but not the global scope.
- **Writes local, reads global.** The eval writes to local variables, but reads from the global scope. This category includes calls to global functions like *Math*.abs().
- **Purely module-local.** The eval reads and writes to outer scopes, but not the global scope. Reconsidering the Point function from Sect. 2, the eval in the case where op=="w" will write the values for x or y into the outer scope.

– **Writes module, reads global.** The eval writes to outer scopes, but not the global scope, and reads from the global scope.
– **Global** The eval reads and writes from and to the global scope. For instance, writing to an undeclared variable will add and/or modify this variable in the global namespace. When the variable window is only defined in the global scope, then using this name for property access renders the side-effect evident.



(a) INTERACTIVE (b) RANDOM

Fig. 6. Scope of eval for INTERACTIVE and RANDOM data sets. (PAGELOAD follows RANDOM)

Fig. 6 shows the scope of operations performed by evals collected in each data set. In terms of writes, the vast majority of evals are actually quite local, not modifying any outer scopes or the global scope. However, reads are more evenly split. It is reasonable to assume that many eval strings are pure, in that they have no side-effects outside the local scope, but not self-contained, as their behavior will nonetheless depend on outer scopes and if it were for using global functions only. Very few eval strings were recorded which affected non-local scopes without affecting the global scope, which suggests there is little modularity in the behavior of eval'd code. Although the majority had relatively local behavior, there is also a significant number of evals with fully global behavior, both reading and writing from and to the global scope. Although some of these are libraries being loaded, manual inspection suggests a number of them are more potentially harmful.

5.2 Patterns

In this section, we discuss each of the common patterns of eval we found. There are many common trends in the use of eval. Some are industry best-practices, such as JSON, asynchronous content and library loading, and code conditional on different JavaScript versions or implementations. Others result from poor understanding of the language,

repeating old mistakes, or adapting to browser bugs. As such, most eval strings can be statically categorized into one of several common patterns. We have identified the following 11 categories:

- **JSON.** As discussed earlier, deserializing JSON is considered to be an acceptable use of eval. New JavaScript engines offer JSON.parse as a safe alternative to eval for JSON, but this function has the caveat that it only accepts perfectly valid JSON, and not near-JSON constructs or JSON assignment, both of which are common in real eval strings. This category includes only strictly-correct **JSON**, as defined by the current ECMAScript standard [6].
- **Relaxed JSON.** This category allows the property initializers to use unquoted variables, allows single quotes anywhere that double quotes are required by JSON, and allows the \x escape in strings. That is, property initializers such as {x:0} and {'x':0} are both accepted, as well as strings such as 'x' and "\x32".
- **=JSON.** This category represents JSON assignment, and accepts assignment expressions in which the left hand side is a variable, and the right hand side is relaxed JSON.
- **Member.** This category covers accesses to a **member** of an object in either the local or global scope. In the vast majority of situations, these evals can be replaced by using JavaScript's hashmap access and by explicitly referencing the global scope if necessary. For instance, if a variable x contains a string which is the name of a global variable, then eval("foo."+ x + "= 3;"); can be replaced by foo[x] = 3;. Concatenations like these are usually simple and repetitive, such as "ebOnScroll", "ebOnResize" and "ebOnKey", all from wordpress.com. This category of eval also often underlies a misunderstanding of JavaScript arrays, such as using eval("subPointArr_"+i) with i as an integer as a substitute for making subPointArr an array (example from zedo.com).
- **Variable.** Another extremely common use of eval is simple variable access, that is, using eval to read or write a variable. One reason why evaling might be beneficial for all code containing variable access results from the scoping rules for eval: As indirect eval always executes in the global scope, evaling code guarantees that all accesses will take place in the global scope instead of the current scope.
- **Variable declaration.** Although substantially less common than simple accesses, a very few sites have been found to use variable declaration within an eval. This is a strange case since it actually modifies the local scope, and can potentially alter the binding of variables *outside* the eval.
- **Typeof.** A case of eval for which we have no satisfying explanation is to eval typeof expressions. For instance, many sites eval strings such as typeof(x)!="undefined". Since eval preserves exceptions, and typeof works with undefined variables whether in eval or not, there is no reason to eval such an expression. typeof in JavaScript is often used to check whether a variable is defined, if(typeof(x)==="undefined")x={}. However, in most cases, this too has clearer alternatives which use JavaScript's hashmap style of field access. For instance, checking for the existence of a global variable can be done more clearly with e.g. if ("x" in window). This misunderstanding can also be combined with a misunderstanding of JavaScript's dynamic objects and hashmap access, such as eval('typeof(zflag_'+y0[i]+')!="undefined"') instead of making zflag a hashmap and using (y0[i] in zflags) (example also from zedo.com).

- **Try/catch.** Another case of eval for which we have no satisfying explanation is to eval try/catch blocks. For instance, bbc.co.uk evals strings such as **try**{**throw** v=14} **catch**(e){}, which is semantically equivalent to v=14 since the throw and the catch parts cancel each other out. Since it's hard to imagine any reason to do this, we can only assume that this code is a strange corner-case of a code generator.
- **Call.** eval is also used to invoke methods only. A common case in this category is document.getElementById, the utility of which is particularly unclear since the parameter to document.getElementById is a string. If only the string parameter varies, then this can be done without eval. If the function called varies, eval can usually be avoided with hashmap syntax as described above. These are usually short and simple, such as document.getElementById("topadsblk01menu") (from sina.com.cn) and update(obj) (from mail.ru). The latter could be done without eval using hashmap style access for the function name, for example window["update'](obj).
- **Library.** Eval is particularly useful for asynchronously loading code. As mentioned earlier, browsers must synchronously parse and execute the code of a <script> element at the moment it is inserted into the page; all other script execution, page rendering, DOM mutation, and even HTML parsing must wait for the script to download and load. Although eval itself has this same blocking behavior, when paired with XMLHttpRequest it can be used to avoid the price of blocking while the script is downloaded. In order to reduce initial page rendering time when loading a website, performance-conscious web developers have taken to embedding a minimal subset of scripts in their main page. Once the user starts reading the rendered page (and no longer cares about latency), remaining scripts are loaded asynchronously via AJAX and eval[8]. Short of a semantic analysis, we consider each string longer than 512 bytes that defines a function to be a library.
- **Other.** On some sites, eval is used with **empty** strings or only **whitespace**, which has no effect. In all the cases we've found of this pattern, the whitespace/empty case has been a default, which is replaced under some circumstances with a string which will have real behavior. The **other** category captures any eval'd string not falling into the previous categories. In particular, it contains method calls interleaved with field access, like foo.bar().zip, but also more complex pieces of code that we did not categorize as a library. As an example consider the following code:

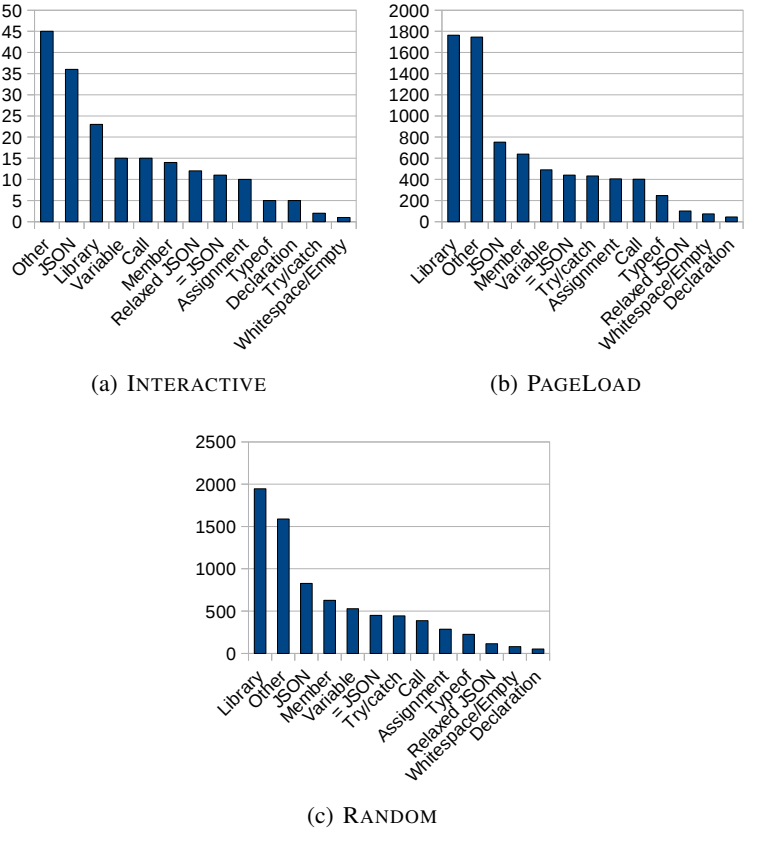
  ```
  var sc_img1 = new Image();
  sc_img1.src = "http://c.statcounter.com/t.php?ip_address=...";
  ```

  which sends data to a server by encoding information into an image's URL in order to circumvent the same-origin policy imposed by the DOM. It is also unclear, however, why this example was eval'd, we speculate that the particular mechanism of circumventing the same-origin policy is determined dynamically and the appropriate one eval'd.

Most eval call sites which evaluate strings in categories other than Library, Other and Variable declaration should be replaceable by less dynamic features such as JSON.parse,

[8] The Doloto project [15] investigated strategies for automating such "code splitting" transformations for network-bound web applications.

hashmap access, and proper use of JavaScript arrays. On INTERACTIVE, these categories account for 83% of all eval'd strings, which suggests that a majority of evals are replaceable. Our further investigation into instances of these categories suggests that often they are sufficiently simple that they could even be replaced automatically.

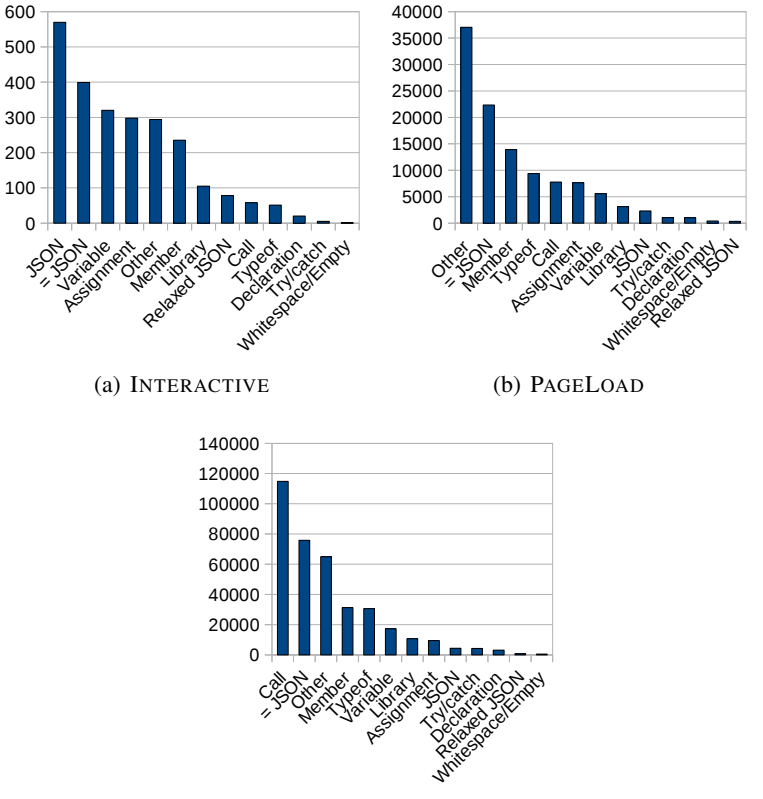


(a) INTERACTIVE (b) PAGELOAD

(c) RANDOM

Fig. 7. Number of web sites in each data set with at least one eval in each category (a single web site can appear in multiple categories).

Fig. 7 shows the number of web sites using evals in each of our categories. The prevalence of uncategorizable evals (other) is quite high, with 45 of the 59 sites in INTERACTIVE which use eval using uncategorizable evals, and nearly half of all sites in PAGELOAD and RANDOM which use eval using eval with uncategorizable strings, at $1763/4063$ and $1589/4309$, respectively. Hand inspection suggests that there is really no unifying category for these, and the actions performed are in fact quite diverse. Fig. 8 shows the number of eval'd strings in each category, as opposed to the number of sites with at least one eval'd string in the category. This view makes it clear that



(a) INTERACTIVE (b) PAGELOAD

(c) RANDOM

Fig. 8. Number of evals in each category.

although uncategorizable evals are used in many sites, we have been able to categorize most strings, with 87.9%, 66.9% and 82.3% of strings categorized for INTERACTIVE, PAGELOAD and RANDOM, respectively.

We see that loading in new functionality through libraries is extremely common amongst the top 10,000 sites, with between 38% (for INTERACTIVE) and 45% (for RANDOM) of all sites which use eval using eval to load new functionality. Fig. 8 indicates that our method of categorizing libraries was in fact quite accurate, as the number of actual evals in this category is quite low, at 3% for both INTERACTIVE and RANDOM and 4% for PAGELOAD. Since most sites will load only a few libraries, and will rarely load more with further interaction, we expect the total number of eval strings in this category to be fairly low, even though its commonality across sites is quite high, and this is verified by the data.

Refreshingly, JSON and variations thereof are in fact quite common. In each data set, JSON or its variations are at worst the second most common category in both Fig. 7 and Fig. 8, and strings in the three JSON categories accounted for between 21% (RAN-

DOM) and 44% (INTERACTIVE) of all strings eval'd. However, this does include the JSON assignment category, which potentially pollutes the scope. Since most call sites do not change categories, as discussed in 5.4, these numbers indicate that analyses could potentially make optimistic assumptions about the use of eval for JSON, but will likely need to accept the caveat of JSON being assigned to a single, often easily-determinable, variable. It seems that the ability to create correct, conformant JSON is restricted primarily to the top 100 sites, with that category accounting for a paltry 1% of strings eval'd in RANDOM, but 23% in INTERACTIVE.

Most of the remaining evals are in the categories of simple accesses, with the exception of Fig. 8 indicating a spike of calls (31% of eval'd strings!) in RANDOM. Fig. 7 suggests that the spike of calls is caused by at most 9% of sites, and it is likely that most of these come from a very few problematic sites, and so this is not indicative of most sites. Member and variable accesses, both simple accesses which generally have no side-effects, are in all data sets amongst the fourth to sixth most common categories for sites to use, accounting collectively for 23%, 17% and 27% in INTERACTIVE, PAGELOAD and RANDOM, respectively.

The most problematic categories, being those with complex side effects such as general assignments and declarations, and those with unconstrained behavior such as calls, are for the most part less common amongst sites, but seem to be used commonly across those sites that use them at all, indicated by their different rankings in Fig. 7 relative to Fig. 8. Bear in mind, however, that many of the uncategorized evals also have problematic behavior.

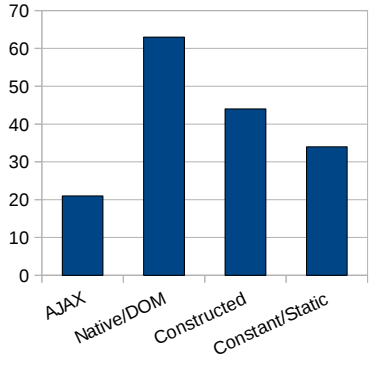
Overall these distributions validate that the industry-standard best practices of JSON and asynchronous library loading are in fact extremely common uses of eval, but the other uses cannot be discounted: they are far from uncommon, and the sites that use them tend to use them quite often, and to perform diverse actions.

5.3 Provenance

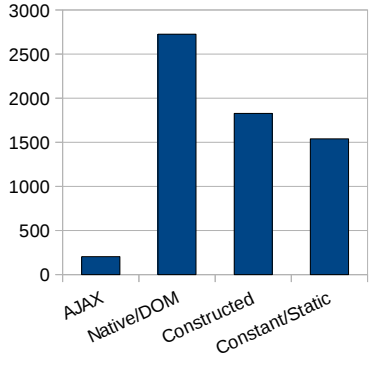
Our infrastructure allowed us to uncover the source of string passed into eval. We group strings according to their provenance in the following categories:

- **AJAX.** All strings constructed from any string retrieved by means of an AJAX call.
- **Native/DOM.** All strings constructed from a string returned by a native method, including strings obtained from the DOM.
- **Constructed.** All strings constructed by concatenating constant strings.
- **Constant.** All strings that appear in the source code file.

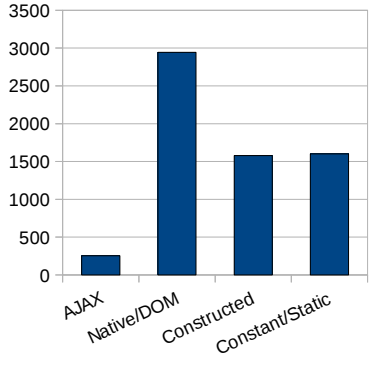
Fig. 9 shows the number of sites that eval strings with a given provenance for our 3 data sets. For the INTERACTIVE case, 20 sites eval strings that were constructed using AJAX data, more than 60 sites eval strings originating in some native method call, more than 40 eval strings constructed from constants, and more than 30 eval pure constants. The proportions for the PAGELOAD and RANDOM sets differ only slightly, both contain less sites evaling AJAX, the RANDOM runs also contain less sites evaling constructed strings in proportion to the other provenances. Fig. 10 shows provenance data for each individual call to eval. It turns out that AJAX is not as popular a technique as we would



(a) INTERACTIVE

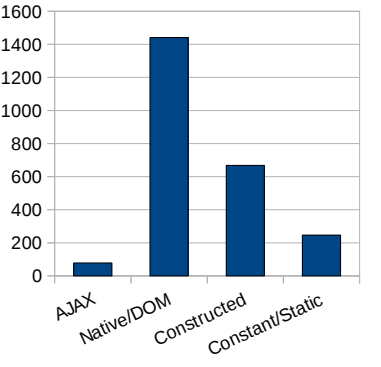


(b) PAGELOAD

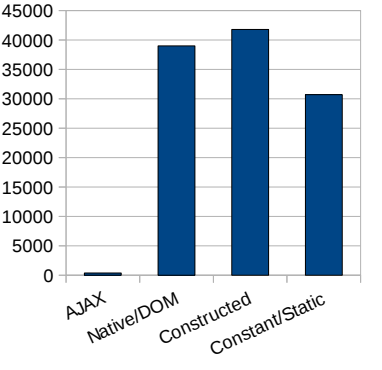


(c) RANDOM

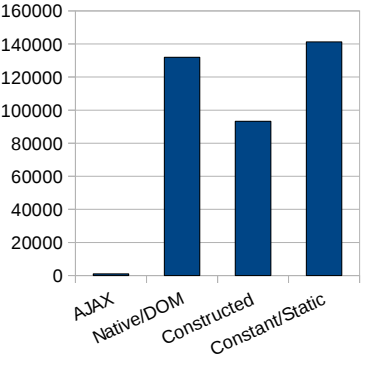
Fig. 9. Number of sites using eval strings of given provenance.



(a) INTERACTIVE



(b) PAGELOAD



(c) RANDOM

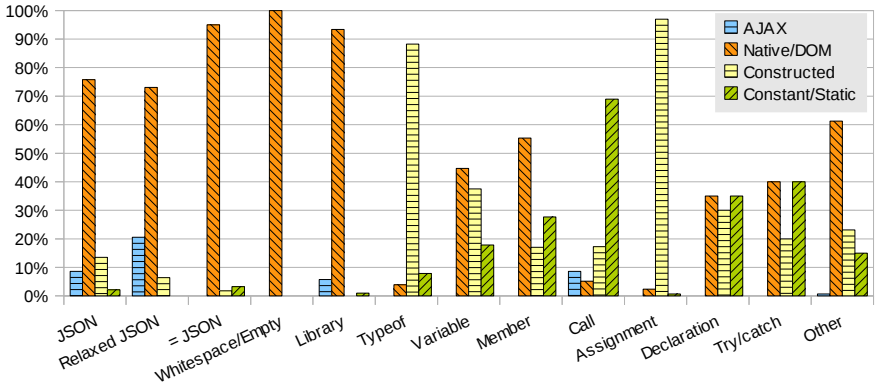
Fig. 10. Provenace of eval strings.

have expected as provenance, in particular for JSON. For the INTERACTIVE runs by far the most dominant source for strings to eval were calls to native methods, including DOM elements. More than 600 strings were constructed from constants only and around 200 strings were just a constant in the source. The distribution of provenance however changes dramatically for the PAGELOAD and RANDOM data sets. For these, Native, Constructed and Constant are roughly equal shares, while AJAX is virtually nonexistent. For the PAGELOAD, there were a little less constants, for the RANDOM runs rather less constructed strings.

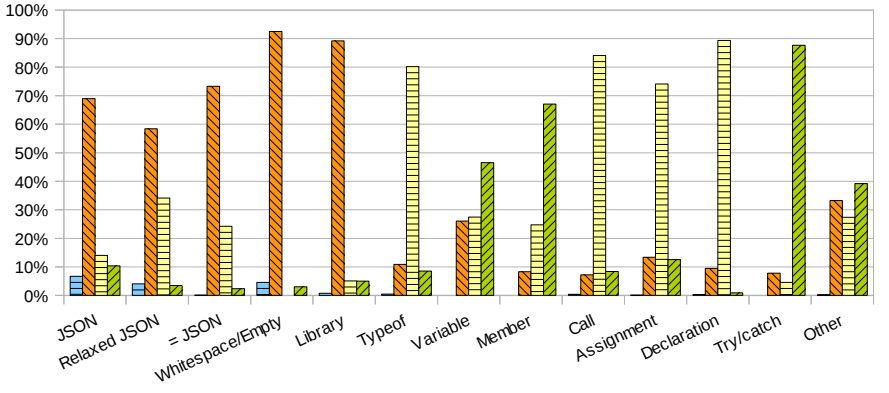
Provenance vs. Patterns Fig. 11 relates the patterns we found with provenance information. We had expected JSON to originate predominantly from AJAX, as this is the standard way of dynamically loading data from one's server. However, for all three data sets the Native/DOM provenance largely outnumbered any other provenance, no matter whether regarding pure, relaxed, or =JSON. The same holds for empty strings and for libraries. When looking into why the proportion of AJAX for these categories is unexpectedly low, we found that for example google.com does retrieve most of its JSON by means of constant values. Detailed investigation yielded that instead of using AJAX, it loads the JSON by means of a dynamically created script tag, which contains a script of the form f("..."). However, instead of using the JSON string directly as a parameter, they parse it with eval in the function f. Therefore, our provenance tracking will categorize this string as a compile time constant. The reason why this site does not use the more standard approach with AJAX is that it stores JavaScript on a separate server where the hostname is a subdomain of google.com. As a consequence, AJAX would not be allowed according to the browser's same-origin policy. Many major web sites have a similar separation of content, and thus must resolve to means other than AJAX to download JSON. We believe that this is the main reason for AJAX being underrepresented. Expressions with typeof were almost exclusively constructed from constants, and therefore not particularly dynamic. However, for the RANDOM runs, a little short of 50% contain results from native method calls, so we assume they contain data provided from a server. From the fact that variables and members are often misused as a replacement for arrays or hashmaps it is not surprising that most of these strings are either constant or constructed from constants. Only in the INTERACTIVE runs did we see a considerable amount of strings that contain data from native method calls. Call is another category with a peculiar distribution: While the INTERACTIVE and RANDOM runs see mainly calls of constants, we intercepted mainly constructed calls during PAGELOAD. We assume that during library installation calls might be constructed, but during interactive phases the majority of methods are statically known. Assignment and Declaration seem to be constructed out of their components, while try/catch blocks are mainly constant, at least for the top 10k sites.

5.4 Consistency

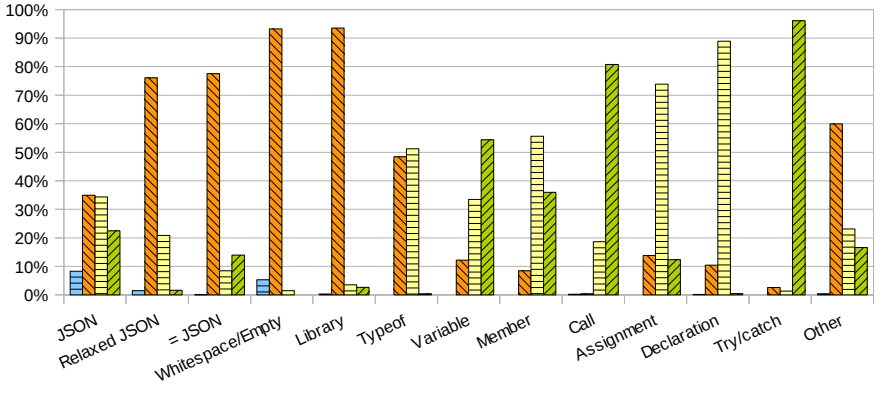
We also analyzed how eval strings evaluated by a single eval callsite changed. Across all of our data sets, we observed only 431 eval callsites with strings in multiple categories (here combining JSON and Relaxed JSON into a single category to avoid false



(a) INTERACTIVE



(b) PAGELOAD



(c) RANDOM

Fig. 11. Distribution of provenances across eval categories in each data set. Y axis is percentage of all evals in that category.

positives). Many of these cases were clearly a single centralized eval used from many branches and for many purposes. For instance, the following three strings are all eval'd by the same callsite.

```
window.location
dw_Inf.get(dw_Inf.ar)
dw_Inf.x0();
```

Several were more perplexing, however, such as a callsite that evals the strings "4", "5" and "a", switching between simple constants and bound variables, and an eval callsite that sometimes evaluated "(null)" (which happens be valid JSON), and other times evaluated "(undefined)" (which is not). Another case evals JSON strings in most cases, but sometimes evaluates JSON-like object literals which include functions, which neither JSON nor relaxed JSON accept.

6 Related Work

Empirical data on real-world usage of language features is generally missing or limited to a small corpus. In previous work, we investigated the dynamic behavior of real-world JavaScript applications [22]. Our results, on a corpus of 103 web sites, confirmed that eval is widely used for a variety of purposes, but we did not scale up our analysis to larger corpus or provide a detailed analysis of eval itself. Ratanaworabhan *et al.* have performed a similar study of JavaScript behavior [21], but their study did not cover eval and focused more on performance and memory behavior. There have been studies of JavaScript's dynamic behavior as it applies to security [26,7] including the role of eval, but the behaviors studied were restricted to those particularly relevant to security and their corpus was rather limited.

Holkner and Harland [11] have conducted a study of the use of dynamic features in the Python programming language, which includes a discussion of eval in that language and similar constructs. Their study concluded that there is a clear phase distinction in Python programs. In their corpus dynamic features occur mostly in the initialization phase of programs and less so during the main computation. Their study detected some uses of eval, but their corpus was relatively small so they could not generalize their observations about uses of eval.

Other languages have facilities similar to eval: for example, reflection in Java can be used to dynamically alter the class hierarchy and to inspect its members. Livshits *et al.* did static analysis of reflection in [16], and Christensen *et al.* [3] analyze the reflection behavior of Java programs to improve analysis precision for their analysis of string expressions.

Several semantics for subsets of JavaScript have been formalized with varying levels of completeness: some are idealized cores that support the formalization of other work [25,4], while others explicitly aim to support most of the language [18,10]. To our knowledge, no formal semantics for JavaScript with support for eval exists. Such a semantics would need to formalize the parsing of JavaScript as part of its own semantics. Without such a formal basis, even analyses which do meaningfully support eval would be very difficult to prove.

7 Conclusion

This paper has provided the first large-scale study of the runtime behavior of JavaScript's eval function. Our study, based on a corpus of the 10,000 most popular sites on the Internet, captures common practices and patterns of web programming. We used an instrumented web browser to gather execution traces, string provenance information, and string inputs to eval.

A number of lessons can be drawn from our study. First and foremost, we confirm that *eval usage is pervasive.* We observed that between 59% of the most popular websites used eval. But luckily, *eval is not necessarily evil.* Loading scripts or data asynchronously with eval is considered a best practice for backwards-compatibility and browser performance, because there is no other facility for asynchronously loading code. We observed that many sites indeed use eval in this manner. While JSON is common, we found that *eval is not used solely for JSON deserialization.* Even if we allowed relaxed JSON notation, this accounts for at most 45% of all calls. Thus, more than half of the calls do in fact use other language features. It seems that *eval is indeed an often misused feature of JavaScript.* While many uses eval were legitimate, many were unnecessary and could be replaced with equivalent but safer code. We found up to 83% of eval uses could be rewritten to use less dynamic language features (builtin JSON parsing function, hashmap syntax, array indexing, the in operator).

Unfortunately for language implementers and designers of static analysis tools, *eval can not be ignored.* Given that 10–20% of side-effects in eval'd code mutate the global object, the effects of eval on globally-scoped variables cannot be ignored by sound static analyses.

Acknowledgments

The authors thank Sylvain Lebresne for his work on the original tracing framework; Keegan Hernandez for his help with recording data and shaking out bugs in our infrastructure; our colleagues at Microsoft, Ben Zorn and Ben Livshits, and Mozilla, Andreas Gal, for support, inspiration and encouragement.

References

1. Christopher Anderson and Sophia Drossopoulou. BabyJ: From object based to class based programming via types. *Electr. Notes Theor. Comput. Sci.*, 82(7), 2003.
2. Christopher Anderson and Paola Giannini. Type checking for JavaScript. *Electr. Notes Theor. Comput. Sci.*, 138(2), 2005.
3. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Conference on Static analysis (SAS)*, 2003.
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.
5. Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2009.

6. European Association for Standardizing Information and Communication Systems (ECMA). *ECMA-262: ECMAScript Language Specification*. Fifth edition, December 2009.
7. Ben Feinstein and Daniel Peck. Caffeine monkey: Automated collection, detection and analysis of malicious JavaScript. In *Black Hat USA 2007*, 2007.
8. S. Guarnieri and Benjamin Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, 2009.
9. Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for ajax intrusion detection. In *International Conference on World Wide Web (WWW)*, 2009.
10. Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 126–150, 2010.
11. Alex Holkner and James Harland. Evaluating the dynamic behaviour of Python applications. In *Australasian Computer Science Conference (ACSC)*, 2009.
12. Dongseok Jang and Kwang-Moo Choe. Points-to analysis for JavaScript. In *Symposium on Applied Computing (SAC)*, 2009.
13. Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Conference on Computer and communications security (CSS*, pages 270–283, 2010.
14. Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Static Analysis Symposium (SAS)*, 2009.
15. Benjamin Livshits and Emre Kiciman. Doloto: code splitting for network-bound web 2.0 applications. In *SIGSOFT FSE*, pages 350–360, 2008.
16. Benjamin Livshits, John Whaley, and Monica Lam. Reflection analysis for Java. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2005.
17. Sergio Maffeis, John Mitchell, and Ankur Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *Computer Security (ESORICS)*, pages 505–522. 2009.
18. Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *Symposium on Programming Languages and Systems (APLAS)*, pages 307–325, 2008.
19. Jan Kasper Martinsen and Hakan Grahn. A comparative evaluation of the execution behavior of javascript benchmarks and real-world web applications (poster). In *Symposium on Computer Performance, Modeling, Measurements and Evaluation (Performance)*, 2010.
20. John McCarthy. History of lisp. In *History of programming languages (HOPL)*, 1978.
21. 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)*, June 2010.
22. Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Programming Language Design and Implementation Conference (PLDI)*, 2010.
23. 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.
24. Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *European Symposium on Programming (ESOP)*, 2005.
25. Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *Symposium on Principles of programming languages (POPL)*, pages 237–249, 2007.
26. Chuan Yue and Haining Wang. Characterizing insecure JavaScript practices on the web. In *World Wide Web Conference (WWW)*, 2009.