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 feature becomes challenging. 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 and 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 presents a large-scale study of the use of eval in JavaScript-based web applications. We have recorded the behavior of 337 MB of strings given as arguments to 550,358 calls to the eval function exercised in over 10,000 web sites. 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.[1] While eval and other dynamic features are a strength of JavaScript, as attested to by their widespread use, their presence is a hindrance to 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 web application developers, but security has proven to be a harder concern to ignore. And, as web applications become central in our daily computing experience, correctness and performance are likely to become more important.

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.[2] This comes with the frequent assertion that eval is the most misused feature of the language.[3] Although eval is a significant feature of JavaScript, it is common for research on JavaScript

[1] While JavaScript provides a few other entry points to code injection, such as setInterval, setTimeout and Function, we refer to this class of features as eval for much of our discussion.

[2] `http://javascript.crockford.com/code.html`

[3] `http://blogs.msdn.com/b/ericlippert/archive/2003/11/01/53329.aspx`

to simply ignore it [2,11,20,1], claim it is hardly used (only in 6% of 8,000 programs in [8]), assume that usage is limited to a relatively innocuous subset of the language such as JSON deserialization and occasional loading of library code [9], or produce a simple warning while ignoring eval's effects [13]. The security literature views eval as a serious threat [19]. Although some systems have unique provisions for eval and integrate it into their analysis [7], most either forbid it completely [15], assume that its inputs must be filtered [5] or wrapped [12], or pair a dynamic analysis of eval with an otherwise static analysis [4].

True Eval. The goal of this study is to thoroughly characterize the real-world use of eval in JavaScript. We wish to quantify 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. For all web page executions, we have obtained behavioral data with the aid of an instrumented JavaScript interpreter. We focus our attention on program source, string inputs to eval and other dynamically created scripts, *provenance* information for those strings, and the operations performed by the eval'd code (such as the scopes of variable reads and writes). Though simply loading a web page may execute 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 user input events to 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.

While we focus on JavaScript, eval is hardly unique to JavaScript. 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 since eval'd code can do almost anything.

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:

- We extend the tracing infrastructure of our previous work[18] to record the provenance of string data and monitor the scope of variable accesses.
- We add tools for automatically loading web sites and generating events.
- We report on traces of a corpus of over 10,000 websites.
- We make available a database summarizing behavioral information, including all input arguments to eval, and other execution statistics.
- We provide the most thorough study of the usage of eval in real-world programs to date.
- We instrumented other means of creating a script at runtime and compare their behavior to eval.

Our tools and data are freely available at:

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

2 The Nature of Eval

JavaScript, which is a variation of the language standardized as ECMAScript [6], is supported by 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. An object is a set of properties that behave like a mutable map from strings to values. Method calls are simulated by applying arguments to a property that evaluates to a closure; this is bound to the callee. The JavaScript object system is extremely flexible, making it difficult to constrain the behavior of any given object. One of the most dynamic features of JavaScript is the eval construct, which parses a string argument as source code and immediately executes it. While there are other means of turning text into code, including the Function constructor, setInterval, setTimeout, and indirect means such as adding <script> nodes to the DOM with document.write, 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 excels at enabling interactive development, and makes it easy to extend programs at runtime. Eval can be traced back to the first days of Lisp [16] where eval provided the first implementation of the language that, until then, was translated by hand to machine code. It has since been included in many programming languages, though often under other names or wrapped inside a structured interface.

The Face of Eval. In JavaScript, eval is a function defined in the global object. When invoked with a single string argument, it parses and executes the argument. It returns the result of the last evaluated expression, or propagates any thrown exception. eval can be invoked in two ways: If it is called directly, the eval'd code has access to all variables lexically in scope. When it is called indirectly through an alias, the eval'd code executes in the global scope [6, sect. 10.4.2]. All other means to create scripts at runtime, as discussed in Sec. 6, execute in the global scope.

The Power of Eval. JavaScript offers little in the way of encapsulation or access control. Thus, code that is run within an eval has the ability to reach widely within the state of the program and make arbitrary changes. An eval can install new libraries, add or remove fields and methods from existing objects, change the prototype hierarchy, or even redefine built-in objects such as Array. To illustrate the power of eval, consider the following example, which implements objects using only functions and local variables.

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

Every invocation of the function bound to 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; any other value updates the 'field'. Calling eval exposes the local scope, thus breaking modularity. Exposing the local scope can be avoided by aliasing eval, but the global scope is still exposed: any assignment to an undeclared variable, such as eval("x=4"), will implicitly declare the variable in the global scope and pollute the global namespace.

Necessary Eval? In modern web applications, the server and client rarely have a persistent connection. Instead, the client makes independent, asynchronous requests every time it needs data. This style of communication is often called Asynchronous JavaScript and XML (AJAX). The data returned is frequently in an application-dependent format, in a portable serialization format such as JSON, or in the form of JavaScript code. If they are in the form of code, then eval is the typical means of evaluating this code. Although the canonical means of making such requests is by using an XMLHttpRequest (XHR) object, it has the drawback that it is subject to the same origin policy, which prevents requests to a different domain. Many sites divide server functions between different hosts, and as such are forced to use other means which are not restricted by the SOP. Most other means actually evaluate the server response as code regardless.

Until recently, JavaScript did not have its own built-in serialization facility, so eval was (and is) often used to deserialize data and code. JSON[4] 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. An example JSON string is:

```
{"Image": {"Title": "View from a Room", "IDs": [11,23,33], "Size": {"Height": 125}}}
```

JSON is restrictive; e.g. {"foo":0} is valid, but {'foo':0} or {foo:0} are not, though all are semantically equivalent JavaScript expressions. Anecdotal evidence suggests that JSON-like strings that don't adhere precisely to the JSON standard are commonly used by developers. JSON is also commonly eval'd along with an assignment to a variable, e.g. x={"foo":0}. Performing the assignment within the eval is unnecessary, as eval returns a result. The canonical way to parse JSON with eval and assign the result to a variable is x=eval(y).

The use of eval is often unnecessary, and is could be replaced by uses of other (less dynamic) features of JavaScript.[5] Consider the following misuse:

```
eval("Resource.message_" + validate(input))
```

The programmer presumably has some Resource object holding a number of messages. To select the right message at runtime, a string such as "Resource.message_error" is built out of some user input. To be on the safe side, the input is validated programmatically. Validation is tricky and a large number of code injection attacks come from faulty validators. The above code could be implemented straightforwardly without eval as Resource["message_"+ input]. Rather than invoking the full power of eval, the code uses a constructed string to index Resources. This achieves the same effect with none of the security risks associated with using eval.

The Eval Within. The eval function is a performance bottleneck because its mere presence affects how a JavaScript engine can optimize and execute surrounding code. Any optimization performed by the virtual machine must account for the black-box behavior of eval. The fact that eval can introduce new variables in the local scope means that flexible, deoptimized bytecode must be generated for a function that contains eval within its body. This version will always be slower than an equivalent function without eval, even if no such variables are actually introduced (see Appendix B).

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

[5] Recent versions of ECMAScript introduced JSON.parse as an alternative to eval.

3 Methodology

We now describe the infrastructure and methodology used to collect our data.

3.1 Infrastructure

We quantify usage of eval by recording relevant information during JavaScript execution, and subsequently performing offline analyses. The data presented in this paper was recorded using TracingSafari [18], an instrumented version of the open-source WebKit project,[6] the common web platform underlying Safari, Google Chrome, and other applications. TracingSafari is able to record low-level, compact JavaScript execution traces; we augmented it to also record properties specific to eval. In particular, we add provenance tracking for strings, as these might eventually become arguments to eval.

TracingSafari records a trace containing most operations performed by the interpreter (reads, writes, deletes, calls, defines, etc.) as well as events for source file loads. Invocations to eval save the string argument. 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 a rich history of the program's behavior, such as access history of each object, call sites, allocation sites, and so on.

3.2 Corpus

Gathering a large corpus of programs is difficult in most languages because accessibility to source code and specific runtime configurations is often limited. On the web, this is generally not the case: any interactive web site uses JavaScript, and JavaScript is only transmitted in source form. Furthermore, most websites are designed to function in many browsers.

JavaScript executes in two distinct phases: first, non-trivial amounts of JavaScript are parsed and executed automatically as the result of loading a document in the browser. Further program execution is event-driven: event handlers are triggered by timers and user input events such as mouse movements, clicks, and the like. To capture a wide range of behavior we have compiled a corpus composed of three data sets:

INTERACTIVE	Manual interaction with web sites.
PAGELOAD	First 30 seconds of execution of a web page.
RANDOM	PAGELOAD with randomly generated events.

All of our runs were based on the most popular web sites according to the `alexa.com` list as of March 3, 2011. INTERACTIVE was generated by manually interacting with the 100 most popular web sites on the Alexa list. Each session was 1 to 5 minutes long and approximated a "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

[6] `http://webkit.org` Rev. 76456.

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. As script execution can recur indefinitely, there is no clear moment when a page has finished loading. 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. One click event is generated per second, for at most 30 events. The final data was recorded between March 3rd and 13th, 2011. All recorded traces are available from our project's site.

These three data sets each cover a useful subset of eval usage in the wild. 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 can generate unrealistic behavior, but is the best way of obtaining a wide variety of behaviors on a large corpus of sites.

3.3 Threats to validity

Program coverage. 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 our corpus is representative of typical browsing behavior. Browser versions are fairly easy to ascertain, so it is possible (and common) for JavaScript code to exhibit behavior peculiar to WebKit. Although this does introduce a subtle bias, all other JavaScript implementations introduce comparable bias.

Diversity of programs. Another threat comes from our focus on client-side web applications. It is likely that other categories of JavaScript applications would display different characteristics. For instance, widgets appear to do so [8]. 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.

4 Usage Metrics

This section presents a high-level picture of the usage of JavaScript and eval in a broad selection of web pages, as summarized in Table 1. At the time of our study, *all* of the top 100 sites used some JavaScript. For the 10,000 most accessed web sites we found that 89% rely on JavaScript. Similarly, eval was used widely and frequently in our corpus. We have recorded 550,358 calls to eval for a total of 337 MB of string data. Over 82% of the top 100 pages use eval, and 50% of the remaining 10,000 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 relying on eval do so even without user interaction. On the other hand, the number of *calls* to eval increases significantly in RANDOM.

Table 1. Eval usage statistics.

Data Set	JavaScript used	eval use	Avg eval (bytes)	Avg eval calls	total eval calls	total eval size (MB)	total JS size (MB)
INTERACTIVE	100%	82%	1,210	84	7,078	8.2	204
PAGELOAD	89%	50%	655	34	158,994	99.3	1,319
RANDOM	89%	52%	627	61	384,286	229.6	1,823

JavaScript code size. As in our previous study, we found that most web sites have less than 512KB of JavaScript code, with some significant outliers, especially in the most popular sites. Fig. 1 displays the distribution of the total size of the JavaScript code loaded during execution of each website, including source loaded via eval. When the same code is loaded multiple times we only took it into account once. The mean sizes are 973KB for INTERACTIVE, 187KB for PAGELOAD, and 270KB for RANDOM. The largest website was yahoo.com with 5.09MB of JavaScript code. The difference in code size between PAGELOAD and RANDOM is explained by the fact that a mouse click (or any other event) may cause additional code to be loaded.

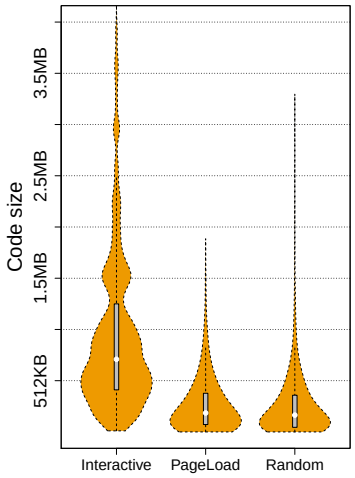


Fig. 1. Code size. The distribution of total size of code loaded during evaluation of each website.

Number of eval *call sites.* We observed that the average number of call sites is small, and interactive behavior is correlated with a greater number of call sites. Fig. 2 shows the distribution of the number of direct call sites to the eval function that are reached per session, for sessions where at least one call to eval was made. User interactions frequently uncovered new call sites: while the mean number of call sites is only 1.7 in PAGELOAD, the mean of RANDOM and INTERACTIVE is 4.0 and 13, respectively. The maximum number of call sites in INTERACTIVE was 77, which is lower than both PAGELOAD and RANDOM (127 and 1331 call sites, respectively).

Number of calls to eval. Unsurprisingly, user interaction is correlated with the number of calls to eval, and websites call eval in both phases of script execution. We observed an average of 38 calls to eval in the INTERACTIVE data set, 28 in PAGELOAD, and 85 in RANDOM. Fig. 3 gives the distribution of the number of invocations of eval per website. The largest number of invocations occurs in RANDOM with a whopping 111,535 calls.

Amount of source loaded by eval. The size of source text passed to eval widely varies depending on *what* is being evaluated. Fig. 4 shows the distribution of source text size. Strings range in size from empty strings to large chunks of data or code. While for INTERACTIVE about two thirds of the strings are less than 64 bytes long, the maximum observed size was 225KB. The PAGELOAD and RANDOM data sets tell similar stories, 85% and 80%, respectively, of strings are less than 64 bytes, but they peak at

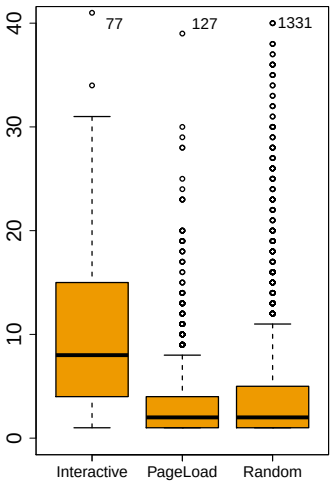


Fig. 2. Eval call sites. The y-axis is the distribution of the number of call sites to the eval function in websites that call the function at least once. (Max value appear on top)

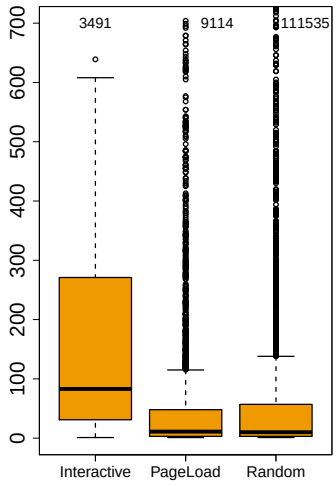


Fig. 3. Eval calls. The y-axis is the distribution of the number of calls to the eval function in websites that call the function at least once.

460KB and 515KB respectively. The average source size is 1,210 bytes for the INTERACTIVE, 655 bytes for the PAGELOAD, and 627 bytes for the RANDOM runs. JSON in particular carries more data on average than other categories. The average size of JSON strings was 3,091 bytes in INTERACTIVE, 2,494 bytes in PAGELOAD and 2,291 bytes in RANDOM. However the medians were considerably lower (1,237, 31 and 54 bytes, respectively), which is consistent with the distribution of sizes seen for other categories. The maximum JSON size is 45KB for INTERACTIVE and 459KB for the other data sets.

Amount of computation via eval. With the exception of loading JavaScript libraries via eval, most calls performed relatively few operations. Fig. 5 shows the distribution of eval trace lengths. The trace length is a rough measure of the amount of computational work performed by any given eval. The operations captured in a trace include object access and update, calls as well as allocation. The median number is again low, 4, with the third quartile reaching 10 operations. The spread beyond the third quartile is extreme, with the RANDOM sessions recording traces of up to 1.4 million operations. Given the maximum size of the source strings passed to eval reported in Fig. 4 this size is not too surprising. In contrast, the maximum number for the INTERACTIVE sessions is low compared to its maximum size of source strings.

In all datasets, the largest eval'd strings, both in terms of length and in terms of event count, were those that loaded libraries. In JavaScript, loading a library is rarely as simple as just installing a few functions; tasks such as browser and engine capability checks, detection of other libraries and API's, creation of major library objects and other such initialization behavior constitutes a large amount of computation relative to other eval calls.

Aliasing of eval. We observed that few programmers took advantage of the differing behavior that results from calling an alias of eval. In INTERACTIVE, 10 of the top 100

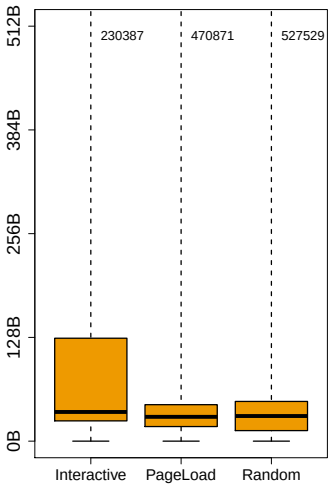


Fig. 4. Eval string sizes. The y-axis is the distribution of the size of eval arguments in bytes.

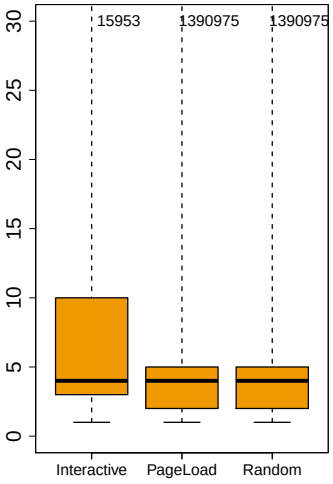


Fig. 5. Events per eval. The y-axis is the distribution of the number of events performed in an eval.

sites aliased eval, but calls to such aliases accounted for only 0.9% of all eval invocations. In PAGELOAD and RANDOM, only 130 and 157 sites, respectively, used used an alias of eval, accounting for 1.3% and 0.8% of eval strings respectively. Manual inspection revealed use cases where programmers used an alias of eval to define a function in the global scope, without realizing that the same effect could be achieved by simply assigning a closure to an undeclared variable. See Appendix C for an illustration.

Presence of JavaScript libraries. In our corpus, JavaScript libraries and frameworks were present on over half of all sites. Table 4 gives the proportion of the sites using common libraries. We found that jQuery, Prototype, and MooTools were used most often. JQuery is by far the most widespread library, appearing in more than half of all websites that use JavaScript. Other common libraries were detected in under 10% of all sites. The Google Closure library used by many Google sites is usually obfuscated, and thus not easily detectable. We do not report on it here. Libraries are sometimes loaded on demand, as shown by the spread between the PAGELOAD and RANDOM (for instance 53% and 60% for JQuery).

One might wonder if libraries are themselves a major contributing factor to the use of eval. Manual code review reveals that eval and its equivalents (the Function constructor, etc) are not required for their operation. The only uses of eval we have discovered are executing script tags from user-provided HTML strings, and as a fallback for browsers lacking JSON.parse. Thus, libraries are not a significant contributor to the behavior or use of eval.

Data Set	jQuery	Prototype	MooTools
INTERACTIVE	54%	11%	7%
PAGELOAD	53%	6%	4%
RANDOM	60%	7%	6%

Table 2. Common libraries. Percentage of website loading one of the following libraries: `jquery.com`, `prototypejs.org`, `mootools.net`. We have no data for `code.google.com/closure`.

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. We look at five important axes. Firstly, we study the **mix** of operations performed by the code executed from an eval. Next, 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. Thirdly, 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. Fourthly, we investigate the **provenance** of the string passed into eval. This comes directly from a desire to better understand 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 Operation Mix

The operations recorded in our traces are simplified, high-level versions of the WebKit interpreter's bytecodes. We report on stores (STORE), reads (READ) and deletes (DELETE) of object properties. These include indexed (x[3]) and hashmap style (x["foo"]) access to object properties. We also report on function definitions (DEFINE), object creations (CREATE), and function calls (CALL). Fig. 6 gives the distribution of operations performed by eval'd code for each of our three data sets. The distribution of operation types across the PAGELOAD data set is consistent with earlier findings, and suggests that eval'd code is not fundamentally different from general JavaScript code. In particular, eval is not solely used for JSON object deserialization, as some related work assumes. That said, INTERACTIVE sessions do contain a greater proportion of 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. 5.3. The RANDOM sessions had a greater proportion of CALL events, likely as part of handling the randomly generated mouse events.

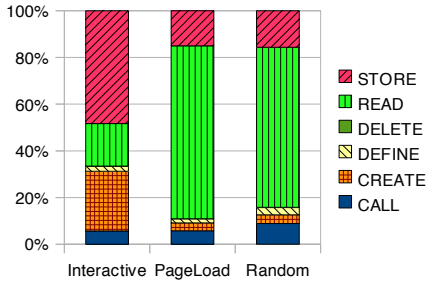


Fig. 6. Operation mix. Proportion of stores, reads, deletes, defines creates and calls performed by eval'd code.

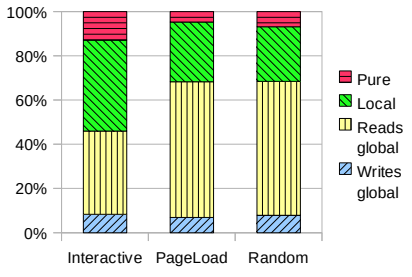


Fig. 7. Scope. Distribution of locality of operations performed by eval.

5.2 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 self-contained and preferred. Our instrumentation determines statically what eval strings have no unbound variables and so are pure, and dynamically logs reads and writes to non-local variables. We categorize the locality of accesses within each call to eval into the following sets.

Pure	Access newly created objects.
Local	Access to local variables and objects.
Read Global	Same as Local + read properties of the global object.
Write Global	Same as Local + read/write/delete properties of the global object.

These categories help us understand the interplay between eval and encapsulation. The **Pure** category captures code that is restricted to creating objects and reading/writing their properties. All JSON code fits in this category. This is the safest category because it neither relies on nor affects the environment of the eval. The **Local** category includes cases where the eval'd code either reads, writes or deletes variables of the function that called the eval (or one of its lexically enclosing functions). The **Read Global** category extends the previous one with the ability to read properties of the global object. Potentially most dangerous is the **Write Global** category, consisting of eval'd code that also can add, modify or delete properties of the global object. 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. Also, in cases when the global object is aliased, resulting global read/writes may be underreported.

We found that evals in the **Pure** and **Writes Global** categories were scarce, while the other two categories were much more common. Fig. 7 shows the scope of operations performed by evals collected in each data set. While the number of **Pure** strings is quite low, the vast majority of evals are actually quite local: only 7 to 8% of all evals modify the global scope for all data sets. However, reads are more evenly split, 38 to 61% of all evals read from the global scope. It is reasonable to assume that many eval strings even in the **Local** and **Reads Global** categories have no side-effects outside the local scope, but are not self-contained, as their behavior will nonetheless depend on the global scope and if it were for using global functions only. Code passed to eval that is neither pure nor global (and so must be designed to work with a particular scope and eval call site) accounts for more than 41% of all eval strings in all data sets.

5.3 Patterns

There are many common patterns in the use of eval. Some are industry best practices, such as JSON, and asynchronous content and library loading. Others result from poor understanding of the language, repetition of old mistakes, or adapting to browser bugs. While it is not possible to be exhaustive, we have nevertheless identified 9 frequently occurring patterns of eval strings which can be detected by a simple syntactic check (a more precise description of how strings are categorized appears in Appendix A):

JSON	A JSON string or variant.
JSONP	A padded JSON string.
Library	One or more function definitions.
Read	Read access to an object's property.
Assign	Assignment to a local variable or object property.
Typeof	Type test expression.
Try	Trivial try/catch block.
Call	Simple function/method call.
Empty	Empty or blank string.
Other	Uncategorized string.

JSON-like constructs. Deserializing JSON is often seen as an acceptable use of eval. The **JSON** category covers strings that are in JSON syntax [6], as well as relaxed notions that permit equivalent JavaScript literals. The **JSONP** (JSON with padding) category covers strings which either assign a JSON expression to a variable or pass a JSON expression as an argument to a function. This pattern is often used for load balancing requests across domains. These other domain names violate the browser's same origin policy, precluding the use of XMLHttpRequest to load JSON from these servers. As a workaround, many standard libraries dynamically create JSONP expressions, typically a function call that takes a JSON argument. The function is a callback function that assigns the JSON data to a variable and processes that data.

Library loading. Libraries can be loaded by <script> tags in the document, but downloading, parsing, and evaluating scripts is synchronous with layout and other events. Blocking ensures deterministic page loading, since scripts can modify the DOM in-place. Although HTML5 introduces new mechanisms for deferred loading, their use is not widespread. A common workaround is to download the script asynchronously with AJAX, then execute it with eval at some later time. This does not block page parsing or rendering immediately, but leaves the programmer the burden of ensuring a known, consistent execution state. The **Library** category attempts to capture this pattern of use. A simple heuristic detects libraries: any eval string that is longer than 512 bytes and defines at least one function. Manual inspection revealed this to be a reasonable heuristic.

Field access. Access to properties of an object and to local variables is covered by the **Read** category. In the vast majority of situations, property reads can be replaced either by using JavaScript's hashmap access or by explicitly referencing the global scope. For instance, eval("foo."+x) can be replaced by foo[x]. Concatenations like these are usually simple and repetitive. This pattern also often underlies a misunderstanding of arrays, such as using eval("subPointArr_"+i) instead of making subPointArr an array. Another common use of eval is variable access. One reason why evaling might be useful comes from the scoping rules for eval. Using an aliased eval guarantees that accesses to variables will occur in the global scope. As mentioned before, this feature found little use. The **Assign** category comprises all statements that assign a value to a variable. A few sites have been found to use variable declarations within an eval. This actually modifies the local scope, and can alter the binding of variables around the eval.

Strange patterns. A strange expression pattern is the category which we call **Typeof** and which covers typeof expressions. For instance, typeof(x)!="undefined". It in not necessary to use eval for this expression. typeof is often used to check whether a variable

is defined and define it if not, 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 if(”x” in window). This misunderstanding can also be combined with a misunderstanding of object access, such as eval(’typeof(zflag_’+y0[i]+’)!=”undefined”’) instead of making zflag a hashmap and using y0[i] in zflags.

Another case for which we have no satisfying explanation, labeled **Try**, is to eval try/catch blocks. For instance, bbc.co.uk evals try{throw v=4}catch(e){} which is semantically equivalent to v=4 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.

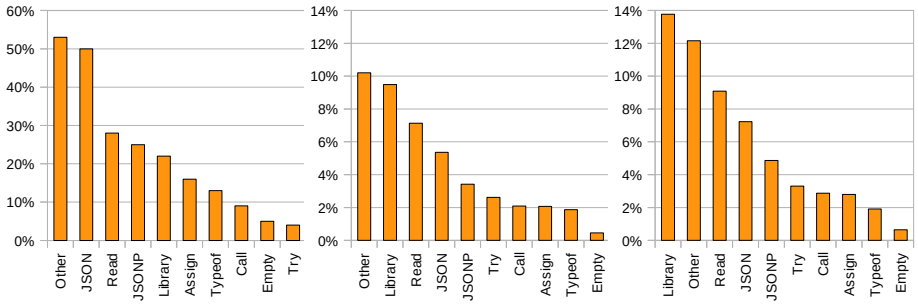
Function invocation. The **Call** category covers evals that invoke methods with parameters that are not padded JSON. A common case in this category is document .getElementById, the utility of which is particularly unclear since the parameter to document.getElementById is a string. If only the string 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”) and update(obj). The latter could be done without eval using hashmap style access for the function name, for example window[”update”](obj).

Other categories. The **Empty** category is made up of empty strings and strings containing only whitespaces. This pattern seems to be the default (empty) case for generated eval strings. Finally, 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:

```
eval(”img1.src=’http://c.statcounter.com/t.php?ip_address=xx’;”);
```

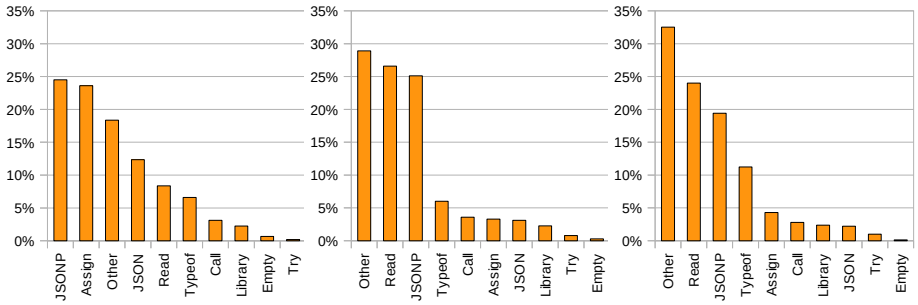
which encodes data into a URL and sends an HTTP GET request in order to circumvent the same origin policy imposed by the DOM. It is also unclear why this example was passed to eval; we speculate that the particular mechanism of circumventing the same-origin policy is determined dynamically and the appropriate one used.

Distribution of categories. Almost all eval categories are present in each data set. Fig. 8 shows the number of *web sites* using each of the eval categories. The prevalence of **Other** evals is high, with 53 sites in INTERACTIVE using uncategorizable evals, 1020 sites in PAGELOAD and 1215 in RANDOM. Manual inspection suggests that there is no unifying category for these, and the actions performed are in fact quite diverse. Fig. 9 shows the number of eval *strings* in each category. Although uncategorizable evals are used in many sites, we have been able to categorize most strings, with 82%, 71% and 67% of strings categorized for INTERACTIVE, PAGELOAD and RANDOM, respectively. We see that loading libraries is common, and between 9% (for PAGELOAD) and 22% (for INTERACTIVE) of sites were detected doing so. Fig. 9 indicates that our method of categorizing libraries is accurate, as the number of actual evals in this category is quite low, at 2% for all data sets. Since most sites load only a few libraries, we expect the total number of eval strings in this category to be low.



(a) INTERACTIVE (b) PAGELOAD (c) RANDOM

Fig. 8. Patterns by websites. Number of web sites in each data set with at least one eval argument in each category (a single web site can appear in multiple categories).



(a) INTERACTIVE (b) PAGELOAD (c) RANDOM

Fig. 9. Patterns. Ratio of evals in each category.

Both JSON and JSONP are quite common. In each data set, JSONP is at worst the third most common category in both Fig. 8 and Fig. 9, and JSON and JSONP strings accounted for between 22% (RANDOM) and 37% (INTERACTIVE) of all strings eval'd. Since most call sites do not change categories (discussed later in Section 5.5) these numbers indicate that analyses could make optimistic assumptions about the use of eval for JSON, but will need to accomodate the common pattern of JSON being assigned to a single, often easily-determinable, variable.

Most of the remaining evals are in the categories of simple accesses. Property and variable accesses, both simple accesses which generally have no side-effects, are in all data sets amongst the second to fifth most common categories for sites to use. They account for 8%, 27% and 24% of eval calls in INTERACTIVE, PAGELOAD and RANDOM, respectively. The most problematic categories[7] appear in fewer sites, but seem to be used frequently in those sites where they do appear. However, this does not include uncategorized evals, which also have problematic and unpredictable behavior.

Impact on analysis. Most eval call sites in categories other than **Library**, **Other** and **Call** are replaceable by less dynamic features such as JSON.parse, hashmap access, and proper use of JavaScript arrays. On INTERACTIVE, these categories account for

[7] By problematic categories, we include evals with complex side effects such as assignments and declarations, and those categories with unconstrained behavior such as calls.

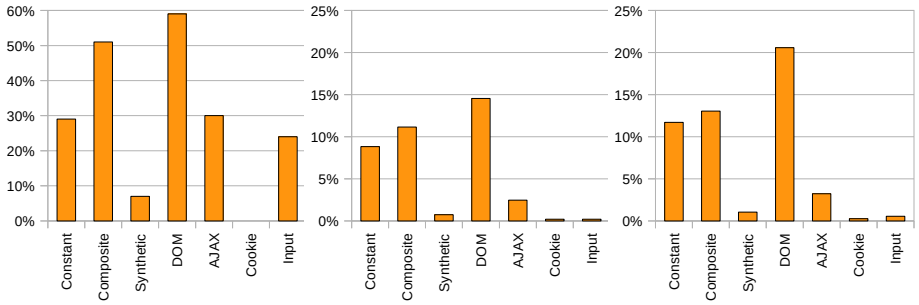
76% of all eval'd strings; thus, a majority of eval uses are not necessary. Upon further investigation into instances of these categories, we believe that they are sufficiently simple to be replaced automatically. While we were able to confirm that best practices of JSON and asynchronous library loading are common uses of eval, 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.

5.4 Provenance

Cross-site scripting attacks (XSS) often make use of eval to run arbitrary JavaScript code. To better understand where eval'd strings come from, we tagged all strings with provenance (tainting) information and instrumented all built-in string operations to preserve provenance information. The return values of certain HTML-specific operations (see below) were also tagged with provenance. We group strings by provenance in the following categories, where later categories may include all previous:

Constant	Strings that appear in the source code.
Composite	String constructed by concatenating constants and primitive values.
Synthetic	Strings that are constants in a nested eval.
DOM	Strings obtained from DOM or native calls.
AJAX	Strings that contain data retrieved from an AJAX call.
Cookies	Strings retrieved from a cookie or other persistent storage.
Input	Strings entered by a user into form elements.

For an example of **Synthetic** strings, consider x=eval(""+document.location.href+""); y= eval(x). The argument to the first eval is from the DOM. However, because the first eval string is in fact a string literal, x is a string. x has **Synthetic** provenance, to distinguish it from string literals appearing in non-eval code (which have **Constant** provenance). The **Constant** category includes string literals, and the **Composite** category includes strings created by concatenating string literals. The **DOM** category includes the result of DOM queries such as document.body.innerHTML) as well as native methods like Date.toLocaleString().



(a) INTERACTIVE (b) PAGELOAD (c) RANDOM

Fig. 10. Provenance by websites. Percentage of web sites using a string of given provenance at least once in an eval expression.

The INTERACTIVE data set had a much higher appearance rate for all provenance types, which is not surprising. Fig. 10 shows the number of sites that pass strings of a given provenance to eval for our 3 data sets. The percentages of the PAGELOAD and RANDOM sets differ only slightly, and both had fewer strings of **AJAX** provenance.

Provenance data tells a more interesting story when aggregated by the provenance of each call to eval; Fig. 11 presents this view. For the INTERACTIVE data set, the dominant provenance of strings was **Composite**. More than 3,000 strings were constructed from composites of only constants and around 600 strings were just a constant in the source. The distribution of provenance is significantly different for the PAGELOAD and RANDOM data sets. For these, **DOM** and **Constant** are used in equal proportion, while **AJAX** is virtually nonexistent.

Provenance vs. Patterns The eval pattern categories from Section 5.3 help to explain some of the surprising provenance data. Fig. 12 relates the patterns we found with provenance information. We had expected most JSON to originate from AJAX, as this is the standard way of dynamically loading data from a server. However, the **DOM** provenance outnumbers all others. The same holds for **Empty** and **Library** patterns. Upon further investigation into the low proportion of **AJAX** provenance, we found that, for example, google.com retrieves most of its JSON as constant values by means of a dynamically-created <script> tag. This script contains code of the form f('{"x":3}'), where the parameter is a string containing a JSON object. However, instead of using the JSON string directly as a parameter (f({"x":3})), they parse the string in the function f using eval. Our provenance tracking will categorize this string as a compile time constant, as it is a constant in the dynamically created script tag. Because google.com stores its JavaScript on a separate subdomain, this convoluted pattern is necessary to circumvent the same-origin policy (under which the straightforward AJAX approach would be forbidden). Many major web sites have a similar separation of content.

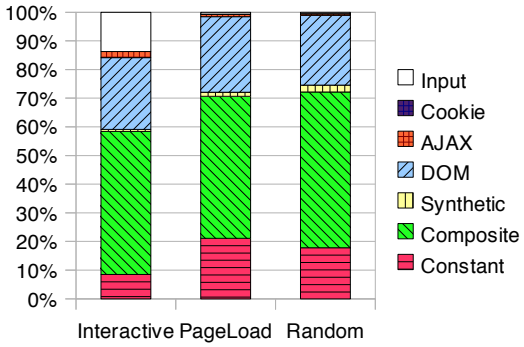
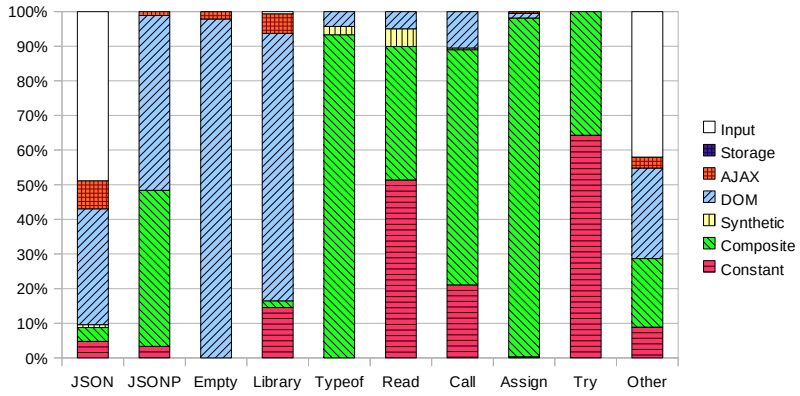
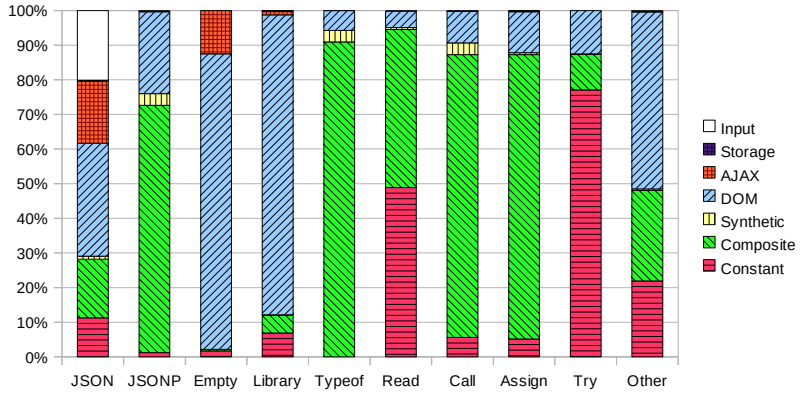


Fig. 11. Provenance. Proportion of strings with given provenance in eval'd strings for the three data sets.

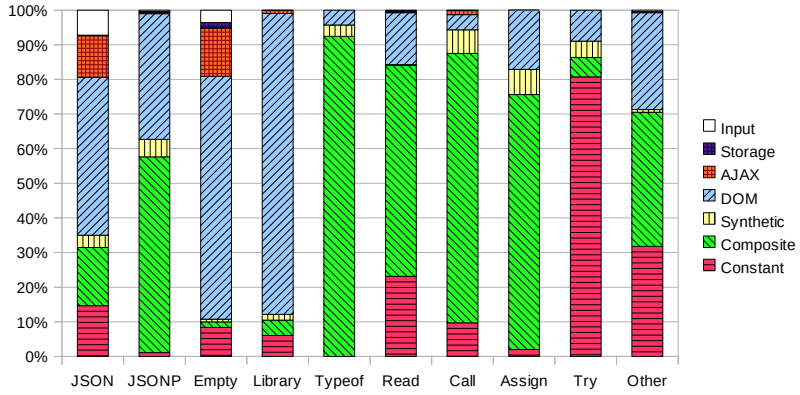
In general, the simpler eval string patterns come from **Constant** and **Composite** sources. Looking at Empty, Typeof, Read, Call, Assign and Try as a group, 85% of these eval'd strings are constant or composite in RANDOM, with similar proportions in the other data sets. Many of these are often misused as replacements for arrays or hashmap syntax, so it is unsurprising that they are generated from constant strings.



(a) INTERACTIVE



(b) PAGELOAD



(c) RANDOM

Fig. 12. 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.

5.5 Consistency

Each eval call site is quite consistent with respect to the pattern of the string argument, but there are exceptions. 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, see Fig. 13. Many of these "polymorphic" 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 call site, found at www.netcarshow.com in RANDOM (although the library that this eval belongs to is found at a few other sites as well). More perplexing call sites include ones that evals the strings "4", "5" and "a", callsites that alternate between simple constants and bound variables, and a call site that at times evaluated "(null)" (which happens be valid JSON) and at other times evaluated "(undefined)" (which is not). Another call site evals JSON strings in most cases, but sometimes evaluates JSON-like object literals which include function literals, which neither JSON nor relaxed JSON accept. Of the 399 eval call sites with strings in multiple patterns, the maximum number of categories was 5, at the call site mentioned above.

Patterns	1	2	3	4	5
Callsites	27553	303	92	3	1

Fig. 13. Consistency. Number of different patterns per call site.

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

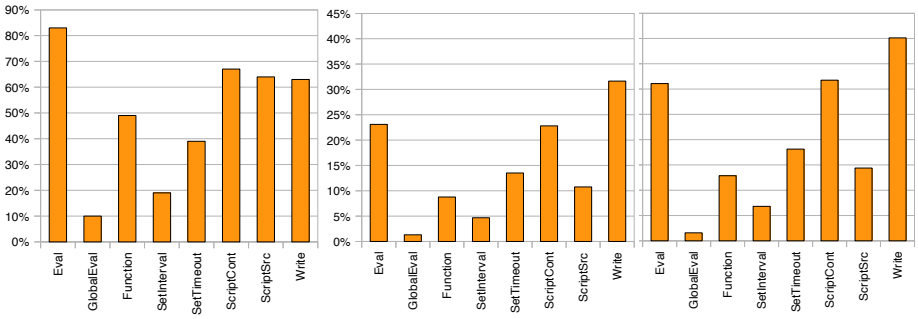
6 Other Faces of Eval

Eval is only one of several entry points to generate executable JavaScript code dynamically. This section reports on the use of the other methods of dynamic code generation available to programmers. We identified the following eight mechanisms of dynamic code generation provided to web programmers:

Eval	Call to eval, executing in local scope.
GlobalEval	Call to an alias executing in global scope.
Function	Create a new function from a pair of strings. (Global scope)
SetInterval	Execute a string periodically. (Global scope)
SetTimeout	Execute a string after a specified point in time. (Global scope)
ScriptCont	DOM operation that changes the contents of a script tag. (Global scope)
ScriptScr	DOM operation that changes the src attribute of a script tag. (Global scope)
Write	DOM operation that writes to the document in place. (Global scope)

The first three mechanisms are part of the JavaScript language. An example is the code var y=Function("x", "print(x)") which creates a new function that takes the parameter x and passes it to the print function. The following two mechanisms are not standardized but commonly implemented as properties of the window object. A simple example is setTimeout("callback()",1000) which invokes the callback function after 1 second. The final three mechanisms are related to DOM[8] manipulation. **ScriptCont** covers changes to script tags such as setting the text or innerHTML property, or calling

[8] The Document Object Model (DOM) represents an HTML page as a tree, where nested tags are encoded as child nodes.



(a) INTERACTIVE (b) PAGELOAD (c) RANDOM

Fig. 14. Dynamic Code Generation by websites. Number of web sites in each data set that are dynamically creating scripts. The x-axis displays the mechanism to create the script, a single web site can appear multiple times.

appendChild(createTextNode(src)), all of which change the source in the script tag. The **ScriptSrc** covers modifications of the src attribute of a script tag, which downloads the given resource and executes its code. The **Write** category covers uses of document.write to manipulate the DOM in-place while executing JavaScript code. (Libraries that contain document.write cannot be loaded asynchronously.) The use of write is discouraged. Consider the the example below, where the first line outputs "<scr" to the document which is concatenated in place with "ipt>" to create a script tag:

```
<script>document.write("<scr");</script>
ipt>alert("this is malicious");</script>
```

The above code is typical of an attack that tries to fool malware filters.

The prevalence of the different mechanisms varies widely among the data sets, especially between the interactively and automatically-gathered data sets. Fig. 14 displays how many sites use each mechanism at least once. In the INTERACTIVE data set, **Eval** is predominant (present in 83% of sites), but in PAGELOAD and RANDOM this mechanism was only used in 23% and 31% of sites, respectively. The use of the global eval variant (**GlobalEval**) is minor (10% of the pages in the INTERACTIVE data set) and even less so in the other data sets (1.3% and 1.6% for PAGELOAD and RANDOM). The **Function** constructor is frequently used by sites in the INTERACTIVE data set (49%), while the other two data sets make only limited use of it (between 8.8% and 13%).

The remaining non-JavaScript mechanisms are used widely. **SetTimeout** is generally used by twice as many sites as **SetInterval**. **SetTimeout** appears in 39% of the sites in INTERACTIVE data set, and for the other data sets between 13% and 18%. Setting the content of a script tag is widespread in the INTERACTIVE data, where 67% of the sites use it, compared to only 23% in the PAGELOAD and 32% in the RANDOM data. Setting the src attribute of a script tag is only widespread in the INTERACTIVE (at 64%) data set, compared to 10–15% in the other data sets. This seems to be a result of the most popular sites using this mechanism to load content from servers on a different domain. Writing script tags to the DOM is popular for all data sets, with 64% of the INTERACTIVE sites doing this, 32% of PAGELOAD and 40% of RANDOM.

When counted by number of actual uses, the **Eval** construct constitutes a clear *minority*; this is worrying, since other code generation mechanisms tend to be overlooked or ignored in the literature. Fig. 15 shows the distribution of the different mechanisms to dynamically create code. For the INTERACTIVE data set, the **Function** constructor was the most commonly used mechanism, despite **Eval** being present in many more sites. Usage of **SetTimeout** is also quite frequent, accounting for more invocations than **Write**, **ScriptSrc**, and **ScriptCont** combined, despite appearing in fewer sites than those mechanisms. This pattern makes sense when one considers that uses of **SetTimeout** frequently recur (in lieu of using **SetInterval**). For the PAGELOAD data set it is interesting to note that **SetTimeout** is used most frequently, **SetInterval** is rarely used, and 7% of scripts written directly to the DOM. This distribution corresponds well with initial setup of the web page, where some tasks are deferred by **SetTimeout**. This is reinforced by the distribution of the RANDOM data. It creates more scripts by means of **Eval**, and is the only data set where **SetInterval** plays a significant role for script creation. We attribute this to the greater dynamism triggered by our random clicking strategy.

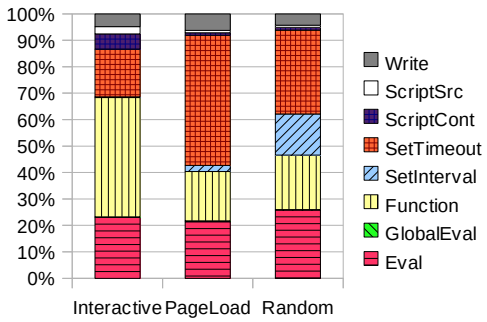
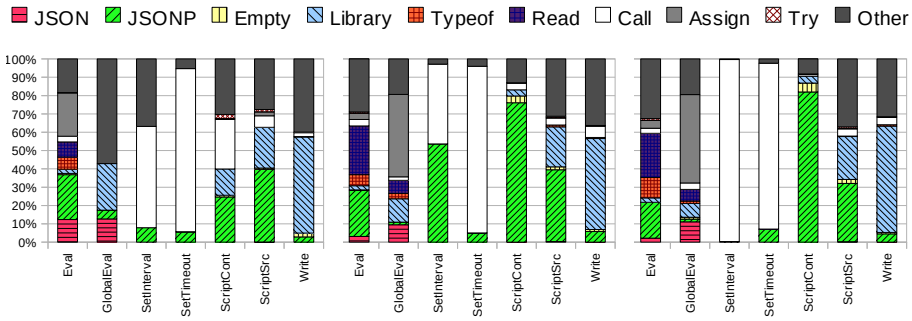


Fig. 15. Dynamic Code Generation. Distribution of mechanisms used to dynamically create scripts per data set.

Classifying the behavior of code created by each of the mechanisms according to the patterns in Sect. 5.3 gives an even better picture of how these mechanisms are commonly used. This classification is depicted in Fig. 16. For all data sets, the local and global **Eval** is used to load the most diverse code, with about 9 significant patterns for these two mechanisms. All the other mechanisms are far less diverse, falling into 3–7 of our defined patterns. setInterval and setTimeout in particular are used almost exclusively with simple function calls (bearing in mind that by JSONP's definition, it is



(a) INTERACTIVE (b) PAGELOAD (c) RANDOM

Fig. 16. Patterns by Dynamic Code Generation. Distribution of code patterns per mechanism of dynamically creating scripts.

often also a simple function call). This consistency suggests that these functions could be speculatively optimized or replaced by safer alternatives in the JavaScript runtime. Even for the other less predictable mechanisms, there is a sufficient lack of diversity that an optimizing compiler could provide faster or safer alternatives according to our patterns.

7 Case Studies

We will now look at individual websites and give examples of their use of eval.

Heute.de: The German news site heute.de (from our RANDOM data set) has a representative example of the naïve use of eval found in many sites, in this case in a snippet which is also found on several other sites. The website contains 49,174 bytes of JavaScript code, with a paltry 136 bytes of eval in 9 calls from the same call site. The eval-using code is summarized in the following snippet:

```
var flashVersion = parse();
flash2Installed = flashVersion == 2;
flash3Installed = flashVersion == 3;
... // same for 4 to 10
flash11Installed = flashVersion == 11;
for (var i = 2; i <= maxVersion; i++)
  if(eval("flash"+i+"Installed")==true)
    actualVersion = i;
```

This example is enlightening in its utter disregard for any consideration of style, legibility and performance. The purpose of the code is to set global variable actualVersion to the version of Flash plugin available in the browser. This is achieved by, first, storing the version number in the local variable flashVersion, then creating 10 new global variables flash$i$Installed (which pollute the namespace and are never used again). Then, to save space perhaps, a loop iterates over an eval that reads a constructed variable name and sets actualVersion. (As an aside, the loop guard, maxVersion, is 10, thus flash11Installed will never be seen.) In this case there is no reason to use eval at all, the entire code snippet could be replaced by the more direct one-liner: actualVersion = parse().

Trainenquiry.com: This Indian train schedule site (also from RANDOM) has 42,135 bytes of JavaScript code and 163 bytes of eval strings across three call sites, all in the ValidatorHookupEvent function (irrelevant code elided):

```
function ValidatorHookupEvent(control, eventType, functionPrefix) {
  var ev;
  eval("ev = control." + eventType + ";");
  eval("control." + eventType + " = func;");
  if (typeof(val.evalfunction) == "string")
    eval("val.evalfunction = " + val.evalfunction + ";");
```

The first two cases are simple misunderstanding of JavaScript which could be expressed more efficiently and succinctly as hash map accesses:

```
ev = control[eventType];
control[eventType] = func;
```

The last one is worth explaining in a little bit more detail. The property val.evalfunction may hold a string, in which case, it is taken to be the name of the function that should

be stored in that property. The conditional will use eval to replace the string with a reference to a function object. This could also have been expressed as

```
if (typeof(val.evalfunction) == "string")
    val.evalfunction = window[val.evalfunction];
```

where eval is again replaced with hash map access to a global property.

Ask.com: Because it loads functionality from several different sources, several of which use eval-equivalent behavior, and it contains a wide variety of behaviors generated by dynamic code, ask.com is an interesting case study. This site loads 2.22MB unique code, 1.39MB of code passed to all variants of eval, and 3.77KB passed to 409 eval calls originating from 48 callsites. The code passed to variants of eval consists of several large libraries from different sources (through <script> tag generation), and two of them, as well as the host, also dynamically generate code. We have excerpted several examples. The site contains ads, and one ad agency's scripts are loaded dynamically by adding <script> tags to the document by means of document.write.

```
document.write("<scr"+"ipt type='text/javascript'
  src='http://afe.specificclick.net/?l=12915&sz=300x250&wr=j&t=j&u="+u
  +"&r="+r+"&rnd="+sm_random+"'></scr"+"ipt>");
```

The behavior of the script added is to save some tracking data, then dynamically load more scripts which themselves load more scripts. This is done by setting the src attribute of a script tag and using document.write.

```
var _comscore = _comscore || []; _comscore.push({ c1: "8", c2: "2101" ,... });
(function() { var s = document.createElement("script"), ...; s.async = true; ...;
document.write("<SCRI"+"PT src='http://ads...."'></SCRI"+"PT>");
```

As a search query is entered, ask.com attempts to auto-complete it. Because auto-completion is performed by a server on a different domain, XMLHttpRequest is not an option and instead a <script> tag is created with the request encoded into the URL. The script loaded in response is a JSONP string. Given the limited portable options for cross-domain communication, this is reasonable.

```
searchSuggestion(["who",["<span ...>who</span> is justin bieber",...]]);
```

An initialization routine is deferred by means of setTimeout with a string argument, presumably to assure that it does not interfere with the loading of the remaining source.

```
setTimeout("JASK.currentTime.init()",JASK.currentTime.SECOND);
```

Since this string is a constant, it could be replaced with a function. We intercepted four different ways to initialize a local variable coming from the same eval call site:

```
function(str){...; eval("var p="+str+";"); return p;}
```

This attempt at JSON deserialization suffers from the dual misconceptions that eval can only evaluate statements and not expressions, and that eval is the only way to deserialize JSON. The eval can be replaced portably by:

```
if("JSON" in window) return JSON.parse(str); else return eval("("+str+")");
```

The reCAPTCHA library updates state in a way that is similar to JSONP, but performs both an assignment and a call, and also uses a relaxed form of JSON. It is loaded similarly to the auto-completion example above.

```
var RecaptchaState = {... timeout : 18000}; Recaptcha.challenge_callback();
```

The following line, which assigns a value to itself, intuitively makes no sense.

```
RichMediaCreative_1298 = RichMediaCreative_1298;
```

This odd behavior is clarified by the original code. A function is loaded with a name containing a unique ID (a timestamp, in fact), and used from other loaded code under that name. Presumably for fear of a miscommunication, eval is used to assure that the created function is assigned to the name that the other code expects.

```
eval("RichMediaCreative_"+plcrInfo_1298.uniqueId+"=RichMediaCreative_1298;");
```

Since the function exists in the global scope, this case is easily replaceable by hashmap syntax over the window object.

8 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 [18]. That result, on a corpus of 103 web sites, confirmed that eval is widely used for a variety of purposes, but in that effort we did not scale up the analysis to a larger corpus or provide a detailed analysis of eval itself. Ratanaworabhan *et al.* have performed a similar study of JavaScript behavior [17] focusing on performance and memory behavior. There have been studies of JavaScript's dynamic behavior as it applies to security [21,7] including the role of eval, but the behaviors studied were restricted to security properties. Holkner and Harland [10] conducted a study of dynamic features in Python, which includes a discussion of eval. Their study concluded that there is a clear phase distinction in Python programs. In their corpus dynamic features occur mostly at initialization 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. Livshits *et al.* did static analysis of Java reflection in [14], and Christensen *et al.* [3] analyze the reflection behavior of Java programs to improve analysis precision for their analysis of string expressions.

9 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 websites 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 50%

and 82% of the most popular websites used eval. Clearly, eval is not necessarily evil. Loading scripts or data asynchronously is considered a best practice for backwards-compatibility and browser performance, because there is no other way to do this. While JSON is common, we found that eval is not used solely for JSON deserialization. Even if we allowed relaxed JSON and JSONP notation, this accounts for at most 37% of all calls. Thus, nearly two thirds of the calls do in fact use other language features. It seems that eval is indeed an often misused feature. While many uses eval were legitimate, many were unnecessary and could be replaced with equivalent and safer code.

We started this work with the hope that it would show that eval can be replaced by other features. Unfortunately our data does not support this conclusion. Removing eval from the language is not in and of itself a solution; eval is a convenient way of providing a range of features that weren't planned for by the language designers. For example, JSON was created to support (de-)serialization of JavaScript objects. It was straightforward to implement with eval, and it is now supported directly in ECMAScript 5. Standards for safer and more consistent library loading have been proposed, e.g. as part of CommonJS. Most accepted uses of eval have been transformed into libraries or new language features recently, and as such no best practices recommends usage of eval. However it is still needed for some use cases such as code generation, which either have not or can not be encapsulated into safer strategies. On the positive side, our categorization was extremely simple, and yet covered the vast majority of eval strings. The categories were chosen to be as restrictive as they are to assure that they are easily replaced by other mechanisms. Restricting ourselves to eval's in which all named variables refer to the global scope, many patterns can be replaced by more disciplined code. The following table illustrates some simple replacements for our patterns

JSON	JSON.parse(str)
JSONP	window[id] = JSON.parse(str) or window[id](JSON.parse(str))
Read	window[id] or window[id][propertyName]
Assign	window[id] = window[id] or window[id][propertyName]=window[id]
Typeof	typeof(window[id]) or id in window
Try	(Not trivially replaceable)
Call	window[id](window[id], ...) or window[id].apply(window, [...])
Empty	undefined or void 0

Furthermore, more than two thirds of the eval strings in these categories listed above are of constant or composite provenance (66.3%, 81.9% and 75.1% in INTERACTIVE, PAGELOAD and RANDOM, respectively) giving a limited number of possible names to be referred to. All but one of these replacements depend on JavaScript's hashmap syntax, which can be used to access properties of objects by string name, but not variables in scope. Since the global scope is also exposed as an object, window, this is sufficient for accessing variables which happen to be in the global scope. However, at least a quarter to a half of eval strings refer to local variables (locality "local": 41.1%, 27.0% and 24.7% in INTERACTIVE, PAGELOAD and RANDOM, respectively; likely everything but "pure"), possibly precluding the use of hashmap syntax. Many of these can be replaced somewhat less trivially in existing code by putting variables which would be accessed by a dynamic name into an object and using hashmap syntax, but for the general case an

extension to JavaScript which would allow to access local variables dynamically would greatly reduce the need for eval.

Acknowledgments. We thank Sylvain Lebresne and Keegan Hernandez for their work on the tracing framework and data recording; as well as Andreas Gal, Shriram Krishnamurthi, Ben Livshits, Peter Thiemann, and Ben Zorn for inspiration and comments. This work was partially supported by a grant from Microsoft Research and by the ONR award N000140910754.

References

1. Anderson, C., Drossopoulou, S.: BabyJ: From Object Based to Class Based Programming via Types. Electr. Notes in Theor. Comput. Sci. 82(7), 53–81 (2003)
2. Anderson, C., Giannini, P.: Type Checking for JavaScript. Electr. Notes Theor. Comput. Sci. 138(2), 37–58 (2005)
3. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer (2003)
4. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged Information Flow for JavaScript. In: Conference on Programming Language Design and Implementation (PLDI). pp. 50–62 (2009)
5. Egele, M., Wurzinger, P., Kruegel, C., Kirda, E.: Defending Browsers against Drive-by Downloads: Mitigating Heap-Spraying Code Injection Attacks. In: Flegel, U., Bruschi, D. (eds.) DIMVA 2009, LNCS, vol. 5587, pp. 88–106. Springer (2009)
6. European Association for Standardizing Information and Communication Systems (ECMA): ECMA-262: ECMAScript Language Specification. Fifth edn. (December 2009)
7. Feinstein, B., Peck, D.: Caffeine Monkey: Automated Collection, Detection and Analysis of Malicious JavaScript. In: Black Hat USA 2007 (2007)
8. Guarnieri, S., Livshits, B.: Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In: USENIX Security Symposium. pp. 151–197 (2009)
9. Guha, A., Krishnamurthi, S., Jim, T.: Using Static Analysis for Ajax Intrusion Detection. In: Conference on World Wide Web (WWW). pp. 561–570 (2009)
10. Holkner, A., Harland, J.: Evaluating the Dynamic Behaviour of Python Applications. In: Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91. pp. 19–28. ACSC '09, Australian Computer Society, Inc., Darlinghurst, Australia (2009)
11. Jang, D., Choe, K.M.: Points-to Analysis for JavaScript. In: Proceedings of the 2009 ACM Symposium on Applied Computing. pp. 1930–1937. SAC '09, ACM (2009)
12. Jang, D., Jhala, R., Lerner, S., Shacham, H.: An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications. In: CCS '10: Proceedings of the 17th ACM Conference on Computer and Communications Security. pp. 270–283. ACM (2010)
13. Jensen, S., Møller, A., Thiemann, P.: Type Analysis for JavaScript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer (2009)
14. Livshits, B., Whaley, J., Lam, M.: Reflection Analysis for Java. In: Yi, K. (ed.) APLAS 2005, LNCS, vol. 3780, pp. 139–160. Springer (November 2005)
15. Maffeis, S., Mitchell, J., Taly, A.: Isolating JavaScript with Filters, Rewriting, and Wrappers. In: Backes, M., Ning, P. (eds.) ESORICS 2009, LNCS, vol. 5789, pp. 505–522. Springer (2009)
16. McCarthy, J.: History of LISP. In: History of programming languages (HOPL) (1978)
17. Ratanaworabhan, P., Livshits, B., Zorn, B.: JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In: USENIX Conference on Web Application Development (WebApps) (Jun 2010)

18. Richards, G., Lebresne, S., Burg, B., Vitek, J.: An Analysis of the Dynamic Behavior of JavaScript Programs. In: Programming Language Design and Implementation Conference (PLDI) (2010)
19. Rieck, K., Krueger, T., Dewald, A.: Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks. In: Annual Computer Security Applications Conference (ACSAC) (2010)
20. Thiemann, P.: Towards a Type System for Analyzing JavaScript Programs. In: Sagiv, M. (ed.) ESOP, LNCS, vol. 3444, pp. 408–422. Springer (2005)
21. Yue, C., Wang, H.: Characterizing Insecure JavaScript Practices on the Web. In: World Wide Web Conference (WWW) (2009)

A Patterns

Patterns are determined in strings by a simple tool itself written in JavaScript, using JSON.parse and various regular expressions. Its algorithm is as follows: With eval string str:

- If str starts and ends with a (and), remove them. This is a common workaround to force certain engines to interpret the string as an expression instead of a statement.
- Strip whitespace from the beginning and end, and comments from any location in the string.
- If JSON.parse(str) does not throw an exception, return **JSON**.
- Relax str into str_relaxed (the relaxation procedure is explained below).
- If JSON.parse(str_relaxed) does not throw an exception, return **JSON**.
- Test str against regular expressions to determine other patterns.

The regular expressions (shown here in the order they are tested) are:

JSONP	/^[A−Za−z0−9_\$\. \t\[\]"']*=[\t\r\n]*(.*);*$/ and matched substring 1 must be JSON or relaxed JSON
Empty	/^$/
Library	/function *[A−Za−z0−9_\$]* *(/ and string must be greater than 512 bytes
Typeof	/^typeof *\(? *[A−Za−z0−9_\$\. \t\[\]"']*\)?$/ or /^typeof *\(? *[A−Za−z0−9_\$\. \t\[\]"']*\)? *[!=<>]+/ or /^if *\(typeof *\(? *[A−Za−z0−9_\$\. \t\[\]"']*\)? *[!=<>]+[^\)]*\)[^\}]*\}? *;? *$/
Read	/^[A−Za−z0−9_\$]*$/ or /^[A−Za−z0−9_\$\.\[\]"']*$/
Call	/^[A−Za−z0−9_\$\.]*\([A−Za−z0−9_\$\.\[\]"', \t\r\n]*\);?$/
Assign	/^[A−Za−z0−9_\$\.\[\]"']* *= *[A−Za−z0−9_\.\[\]"']*;?[\t\r\n]*$/ or /^var [A−Za−z0−9_\$]* *(= *[A−Za−z0−9_\$\.\[\]"']*)?;?$/
Try	/^try *\{[^\}]*\} *catch *\([^\)]*\) *\{[^\}]*\} *;?$/

All other strings are categorized as Other.

The relaxation procedure is a simple process that replaces most JSON-like strings with strict JSON strings. Single-quoted strings are replaced with double-quoted strings (e.g. {'foo':0} becomes {"foo":0}, unquoted names are quoted ({foo:0} becomes {"foo":0}) and a form of string escapes not accepted by JSON (\x) are replaced by their JSON equivalent (\u).

B Performance impact of Eval

The WebKit JavaScript engine will generate different bytecodes for local variable access when a function calls eval.

Consider the two functions in Fig. 17. Because of the presence of eval, the translation of function E() must do dynamic, by-name lookup of x (opcodes resolve_with_base, put_by_id, resolve), whereas NoE() simply refers to statically-known global offsets (opcodes get_global_var, put_global_var). This is a direct example of the potential impact of eval on performance as the code on the left will run slower in a WebKit.

```
function E() {
  eval(evalstr); x++;
  return x;
}
```

```
enter
init_lazy_reg r0
init_lazy_reg r2
init_lazy_reg r1
create_activation r0
resolve_with_base r4, r3,
                  eval(@id0)
resolve r5, evalstr(@id1)
call_eval r3, 2, 12
op_call_put_result r3
resolve_with_base r4, r3, x(@id2)
pre_inc  r3
put_by_id r4, x(@id2), r3
resolve r3, x(@id2)
tear_off_activation r0, r2
ret r3
```

```
function NoE() {
  id(evalstr);
  x++;
  return x;
}
```

```
enter
get_global_var r0, −8
mov r1, undefined(@k0)
get_global_var r2, −12
call r0, 2, 9
get_global_var r0, −11
pre_inc r0
put_global_var −11, r0
get_global_var r0, −11
ret r0
```

Fig. 17. Bytecode generated by WebKit.

C Local vs. Global Scope

The eval function provides two modi operandi. Called directly, it executes in the local scope and only variables that are not declared in that scope will bind to the local scope. However, if called through an alias, then eval executes in the global scopes and all variables, declared or undeclared in the eval string, bind to the global scope. In the following program the first eval executes in the local scope and thus assigns to the local variable x, while the call to the alias of eval assigns to the global variable x.

```
1 (function() {          // the anonymous function creates its local scope
2   var x = eval("x = 4");       // assigns 4 to the local variable x twice
3   var e = eval;        // alias eval to call it in the global scope
4   x = e("x = 4");})()          // first assigns 4 to the global variable x and then to the local variable
```