An Empirical Study on the Rewritability of the `with` Statement in JavaScript

Changhee Park
KAIST
changhee.park@kaist.ac.kr

Hongki Lee
KAIST
petitkan@kaist.ac.kr

Sukyoung Ryu
KAIST
sryu.cs@kaist.ac.kr

Abstract

As JavaScript gets more popular in web programming, a demand for better analysis of JavaScript programs becomes higher. However, many dynamic features of JavaScript make its analysis, especially static analysis, particularly difficult. One of the main dynamic features of JavaScript is the `with` statement, which invalidates lexical scoping by introducing a new scope at run time. To simplify the problem, many researchers leave the `with` statement out of their consideration and major web service companies force their web application developers to use sub-languages of JavaScript which do not include the `with` statement. While deprecating a potentially dangerous feature might be the easiest solution, it could be too restrictive for application development. To decide whether to include the `with` statement, we should better understand the actual usage patterns of the `with` statement.

In this paper, we present the usage patterns of the `with` statement in real-world JavaScript applications currently used in the 98 most popular web sites. We investigate whether we can rewrite the `with` statements in each pattern to other statements not using `with`. We show that we can rewrite all the static occurrences of the `with` statement which does not have any dynamic code generating functions. Even though the rewriting process is not applicable to any dynamically instrumented `with` statements either, our result is still promising. Because all the static approaches that disallow the `with` statement also disallow dynamic code instrumentation, such static approaches can allow the `with` statement using our rewriting process: JavaScript developers use the `with` statement in their applications and the rewriting process desugars it away. Automating the rewriting process is under progress and we believe that it will make static analysis of JavaScript programs more feasible while imposing less restriction.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Experimentation

Keywords JavaScript, `with` statements, rewritability, static analysis

FOOL '11 October 23, 2011, Portland, Oregon, USA.

1. Introduction

JavaScript is the most dominant language in developing web applications. According to Richards *et al.* [17], all of the 100 most popular web sites used JavaScript and among the 10,000 sites, 89% of them used it. We expect that the figure will increase. JavaScript has many dynamic features to provide users with convenient ways to change program states at run time such as adding or deleting members of objects, changing object hierarchies, dynamically instrumented code execution using the `eval` function, and dynamic scope introduction using the `with` statement. This flexibility has brought JavaScript great success in the web industry.

However, the flexibility makes it difficult to predict the behavior of a JavaScript program, which often leads to an error-prone program or a target of security attacks. The vulnerability of JavaScript applications [12] and analysis mechanisms [14, 15] has been reported, and a better analysis of JavaScript applications for more reliable programs has become more important. In consequence, some researchers recently proposed type systems [1, 11, 19], static analyses [9], and combinations of static and dynamic analyses [2, 14] for JavaScript. Also, web service companies constrain web application developers from using specific dynamic features of JavaScript for better analysis [3, 5, 7].

While understanding actual behaviors of JavaScript programs is critical to support better analysis, empirical studies on the usage patterns of dynamic features in JavaScript have not yet been performed extensively. Richards *et al.* [17, 18] recently conducted a large-scale survey on dynamic behaviors of JavaScript applications and the usage patterns of the notorious `eval` function. They provide empirical data to show or, rather, falsify the common assumptions found in literature such as that `eval` is hardly used or that deleting object members rarely happens. We believe that their empirical data fertilize the JavaScript research by supporting reasonable assumptions, dismissing unrealistic assumptions, and stimulating new directions of research.

Inspired by Richards *et al.*, we perform an empirical study on the usage patterns of the `with` statement, which is another dangerous dynamic feature of JavaScript. Even though the `with` statement has been considered harmful, its actual behavior has not been studied yet. Most researchers [1, 8, 11, 19] leave out the `with` statement in their consideration, and some companies [3, 5, 7] require web application developers use only their sub-languages of JavaScript, which do not include the `with` statement. Even for the work that include the `with` statement [10, 14], they do not support the full generality of the `with` statement.

One of our goals in this paper is to provide empirical data for `with` statements used in real-world JavaScript applications. We have collected JavaScript code from the 98 most popular web sites[1], and analyzed the code to show how the `with` statements are actually being used in practice. This data will help reflect the real-world usage of JavaScript in designing and developing JavaScript program analyses. Also, we categorize the usage of `with` statements into seven patterns and check whether we can rewrite the occurrences of `with` in each usage pattern into other code without using `with`. The rewritability is important because if the rewriting is possible, we can apply many static analyses that are only applicable to programs without `with` to more JavaScript programs that use `with`. We are currently working on automating the rewriting process.

In this paper, we present various aspects of the `with` statement. First, we give an overview of `with` including its syntax, semantics, main usage pattern, and some issues in Section 2. We present our analysis result of the usage patterns of `with` in real-world applications in Section 3. In Section 4, we describe our rewriting strategy to remove the occurrences of the `with` statement and check the rewritability of `with` in each usage pattern. We discuss the related work in Section 5, and present our future work and conclude in Section 6.

2. The `with` Statement in JavaScript

In this section, we give an informal description of the syntax and semantics of the `with` statement, explore the good parts of it in the sense of programming convenience and the bad parts in the sense of performance and static analysis, and provide a brief description of its rewritability, which serves a key role in this paper.

2.1 Syntax and Semantics

JavaScript is an implementation of the ECMAScript language standard [6], which defines the syntax and semantics of the language. The syntax of the `with` statement is as follows:

`with(`*exp*`)` *stmts*

where *exp* is an expression and *stmts* is either a single statement or a list of statements enclosed by curly braces. We call *stmts* "the body" of the `with` statement.

The semantics of the `with` statement is as follows: first, *exp* evaluates to a JavaScript object and the object is added to the front of the current scope chain. For convenience, we call the evaluated scope object given to the `with` statement "the `with` object" throughout the paper. Then, all the properties[2] in the `with` object become local variables in the body of the `with` statement. After evaluating the body of the `with` statement, the current scope chain removes the added `with` object and reverts to the original scope chain before evaluating the `with` statement.

While JavaScript has (mostly) lexical scoping using first-class, lexically-scoped functions, the `with` statement invalidates lexical scoping by introducing a new scope at run time. This dynamic nature of the `with` statement often provides flexibility to programmers but it incurs a performance overhead and makes static analyses infeasible. In the following subsections, we first explain the main usage pattern and the usefulness of the `with` statement, and subsequently explain its harmfulness with some examples.

2.2 The Original Intention

As Crockford describes in his blog posting [4], the original intention of the `with` statement is to provide a convenient way to develop dynamically changing web contents in JavaScript. Consider the following simple example:

```
document.body.children[0].style.textAlign="center";
document.body.children[0].style.fontSize=50;
```

The example shows a common usage pattern in JavaScript code found in many web sites to dynamically manipulate DOM[3] objects. The `document` represents the whole HTML document containing the JavaScript code and we can access the DOM object represented by the first tag in the `body` tag of the HTML document using `document.body.children[0]`. By changing the values of the various fields in the `style` attribute of the DOM object, we can change the display format of the contents within the corresponding tag of the HTML document. The example code aligns the displayed text at the center and changes its font size to 50.

The example shows that accessing multiple fields in an object is a common practice in JavaScript and the long field access syntax is not convenient. For example, if we want to change the display format further by changing the `width` and `height` fields in the `style` attribute, we have to either type the long object name, `document.body.children[0].style`, or copy-and-paste it repeatedly, which is tedious and error prone.

In contrast, if we use the `with` statement, we can avoid using such a long text as follows:

```
with(document.body.children[0].style) {
  textAlign="center";
  fontSize=50;
  width=200;
  height=400;
}
```

Instead of using the long object accesses multiple times, the `with` statement enables us to use just the property names in the body of the `with` statement by specifying the `with` object just once in parentheses. By doing so, we can simplify the code and improve its readability.

2.3 Deprecation of the `with` Statement

While the ECMAScript 5th edition [6] still includes the `with` statement for backward compatibility with ECMAScript 3, it forbids `with` in *strict* mode. Whether to include the `with` statement in the next version of ECMAScript is controversial, but its deprecators have two main reasons.

The first reason is the performance overhead caused by a new scope introduction and additional scope lookups for variable references in the body of the `with` statement. Inspired by Resig and Bibeault's small experiment in their book [16], we conducted a similar experiment to show the performance overhead of the `with` statement. Figure 1 presents three code blocks used in the experiment. Both `codeblock1` and `codeblock2` assign the value of the `prop1` property of the `testobj` object to the global variable, `globalvar1`, in the body of the `for` statement. While `codeblock1` uses the dot notation to access the `prop1` property, `codeblock2` uses the `with` statement which introduces the `testobj` object as a new scope object. Similarly to `codeblock2`, `codeblock3` uses the `with` statement but, instead of accessing any property of the `testobj` object, it accesses a global variable, `globalvar2`. With `codeblock3`, we investigate how an additional non-local variable access in the `with` statement affects its performance.

For each code block, we measured the execution times of its 1000 runs in four main browsers. Table 1 presents the averaged results. Comparisons between the execution times of `codeblock1` and `codeblock2` show that using the `with` statement leads to

[1] Among the top 100 web sites, two sites are not accessible directly. We discuss them in more detail in Section 3.1.

[2] In JavaScript, any field or method of an object is called a property of the object.

[3] DOM is a shorthand for Document Object Model which represents an HTML document as a tree form. For more information, see `http://www.w3.org/DOM/`.

```
var globalvar1, globalvar2=3, testobj={prop1 : 3};

//codeblock1
for(var i=0; i<1000000; i++) {
      globalvar1=testobj.prop1;
}

//codeblock2
with(testobj) {
   for(var i=0; i<1000000; i++) {
      globalvar1=prop1;
   }
}

//codeblock3
with(testobj) {
   for(var i=0; i<1000000; i++) {
      globalvar1=globalvar2;
   }
}
```

Figure 1. Code blocks to show the performance overhead by the `with` statement

(unit : ms)

	codeblock1	codeblock2	codeblock3
Chrome 12	1.414	790.399	928.694
Firefox 5	6.599	601.566	694.188
Internet Explorer 9	5.072	118.549	129.509
Safari 5	4.523	299.044	369.203

Table 1. Performance overhead of the `with` statement

```
function fun1(obj) {
  var localvar1=0;
  with(obj)
    localvar1=one;
}

var obj1={one:1}, obj2={two:2};
var obj3={localvar1:3, one:1};

fun1(obj1);
fun1(obj2);
fun1(obj3);
```

Figure 2. A JavaScript program with the `with` statement

a considerable performance overhead in all four browsers; the maximum overhead is the case of Chrome 12 which has more than 500% performance overhead. Also, comparisons between the execution times of `codeblock2` and `codeblock3` show that using an additional non-local variable access within the `with` statement leads to some performance overhead; the maximum overhead is approximately 23% performance overhead in the case of Safari 5. Considering that the performance often takes precedence over correctness or reliability of a program in the web 2.0 era which has many dynamic contents with various user interactions, excessive uses of `with` are harmful.

The second reason is that the `with` statement makes static analysis particularly difficult. Consider a JavaScript program in Figure 2. The `fun1` function takes one parameter, `obj`, and assigns the value of the variable `one` to `localvar1` within the `with` statement under the new scope introduced by it. In the program, the `fun1` function is called three times with three different argument objects, `obj1`, `obj2`, and `obj3`, respectively, and shows a different behavior for each function call. At the first call, the argument `obj1` has the `one` property, which becomes a local variable within the `with` statement of the `fun1` function. Hence, the local variable `localvar1` in `fun1` gets the value of the `one` property in the `obj1` object. At the second call, however, accessing the variable `one` within the `with` statement makes a run-time error since the argument `obj2` does not have the `one` property and accessing undeclared variables is a run-time error in JavaScript. At the third call, the argument `obj3` has a property `localvar1` which has the same name as the local variable in `fun1`. In this case, the name `localvar1` in the body of the `with` statement refers to the property of the `obj3` object and thus the value of the `localvar1` property of `obj3` gets changed as the result of the function call.

To see how the `with` statements make it difficult to statically analyze JavaScript programs, assume that we want to signal undeclared variable accesses in Figure 2 by statically analyzing the program. If we have only lexical scoping, we can easily detect that the access to the variable `one` in the `fun1` function is an illegal access to an undeclared variable. However, as explained above, the variable access can be legal or illegal in each function call depending on the argument of the function call. In this case, for a static analyzer to give the correct answer, it has to check every function call keeping track of its argument object, which makes the analysis complicated or sometimes impossible. As another example of static analysis, assume that we want to guarantee that no JavaScript programs modify the values of some specific variables for security. However, as shown with the third call of the `fun1` function, modified variables in a `with` statement would be different for each function call and it is impossible to figure out which variables are modified within the `with` statement just by looking at the code, unless we completely know which properties are in the `with` object. Therefore, `with` statements are usually deprecated in static analyses.

2.4 Rewritability of the `with` Statement

Given the reasons to deprecate the `with` statement as explained in Section 2.3 and the fact that the commercially used sub-languages of JavaScript [3, 5, 7] disallow the `with` statement, it is natural to ask the following question: is it possible to rewrite the `with` statement to other language constructs? If it is possible, it would be unnecessary to worry about the problems caused by the `with` statement and we have no more reasons to disallow the `with` statement in any sub-languages of JavaScript.

In fact, if we use the `with` statement only for its originally intended uses, which is to avoid repeated occurrences of long object accesses as explained in Section 2.2, obviously we can rewrite the `with` statement to other language constructs. To illustrate this, we revisit the code example using the `with` statement in Section 2.2:

```
with(document.body.children[0].style) {
  textAlign="center";
  fontSize=50;
  width=200;
  height=400;
}
```

This code can be rewritten to the following semantically equivalent code:

```
var v=document.body.children[0].style;
v.textAlign="center";
v.fontSize=50;
v.width=200;
v.height=400;
```

which does not use the `with` statement but uses an additional variable declaration and the explicit field access syntax. While preserving the semantics, programmers can still avoid repeated occurrences of long object accesses without using the `with` statement.

While the above rewriting example is an easy case, a general rewriting rule is slightly more complex. In the above example, the rewriting is easy because we know that all the variables within the `with` statement are the properties of the `with` object. However, in general, we cannot statically know whether a variable in `with` is a property of the `with` object, especially when the `with` object is a function argument or a user input. For a general case, we should inspect whether a variable is a property of the `with` object. Hence, the general form of the `with` statement as follows (we omit unimportant parts by ellipsis):

```
with(obj) {
  ... var1 ...
}
```

where `obj` is the `with` object and `var1` is a variable, is rewritten to the following code:

```
...
if(obj.hasOwnProperty("var1"))
  obj.var1;
else var1;
...
```

where `obj.hasOwnProperty("var1")` returns true if `obj` has the property named `var1` and returns false, otherwise.

With this observation of the `with` rewritability, our research started from the following question:

> "Are there any usage patterns of the `with` statement which we cannot rewrite to other constructs without using `with`?"

We were curious that whether we can eliminate all the occurrences of the `with` statement in real-world applications by rewriting them. To answer such questions, we present the real-world usage patterns of the `with` statement in next sections and extend the discussion on the rewritability of the `with` statement in depth to address the code examples used in practice.

3. Analysis on Real-World Usage Patterns of the `with` Statement

To understand the real-world usage patterns of the `with` statement, we need a data set of JavaScript code used in practice and a mechanism to manipulate the data set so that we can extract necessary information from it. In this section, we describe the methodology of our survey on `with` statements and give the survey result answering the questions such as how much and in which patterns they are used in reality.

3.1 Methodology

JavaScript is mainly used for development of web applications and dynamic manipulation of HTML documents. Therefore, many web sites have embedded JavaScript code in their web pages and we believe that a set of embedded JavaScript code in popular web sites is a reasonable representative of JavaScript applications in practice. Hence, inspired by Richards *et al.* [17], we collected JavaScript code from the 98 most popular sites using their tool, TracingSafari. We describe our methodology in more detail as follows:

Data sets. A web information company, Alexa[4], provides a list of ranked web sites by its own ranking method based on the number of visitors in the web sites. We used a list of the 100 worldwide most popular web sites as of July 14, 2011, provided by Alexa, which includes popular search engine sites such as `google.com` and `yahoo.com`, social networking service sites such as `facebook.com` and `twitter.com`, a video sharing site, `youtube.com`, and other famous web sites. Among the 100 sites, we could not access to two sites: `googleusercontent.com` and `bp.blogspot.com`. We believe that they represent the web sites of their subdomains, and the internet traffic from the subdomains is reflected in ranking the host domains. For example, the `bp.blogspot.com` data presents the data from its subdomains such as `3.bp.blogspot.com`. In this case, it is hard to figure out the complete list of the subdomains of a host domain. Even when it is possible, whether to consider the `with` statements in subdomains as the ones in the subdomains or as the ones in the host domain is not clear and may not be fair to the other 98 directly accessible sites. Hence, we omitted the inaccessible two sites from our list and investigated the remaining 98 sites.

We collected the JavaScript code from the 98 sites in two modes, generating two sets of code data. In the first mode, we visit each site in the list and collect the JavaScript code that is executed for 30 seconds when loading the initial page of the site. This data set represents the JavaScript code used for initial setups of web sites and helps us to characterize the roles of the `with` statement at loading time. In the second mode, in addition to the code in the first mode, we collect the JavaScript code that is executed by user interactions. Because web sites contain fair amount of event-driven JavaScript code in their web pages, the code executed by user interactions provide better coverage of JavaScript code. Note that providing events to each site in a consistent way is a difficult problem because typical events in web sites are very different from sites to sites. To be consistent in all sites, we provide only mouse events, mainly clicking any contents of a web site, which are the most typical events and the most common actions any users can perform in any sites. For the data set, we visit each site and collect the executed JavaScript code while giving mouse events for 2 minutes.

Throughout the paper, we refer the data set from the first mode as `LOADING` and the one from the second mode as `INTERACTION` for convenience.

Tools. We use the tool, TracingSafari, which Richards *et al.* [17] developed and made it available to public, to collect the JavaScript code in each site. TracingSafari is a customized version of WebKit[5], an open-source web browser engine used in Safari. When we visit a web site via the browser provided by TracingSafari, an embedded JavaScript code in the web page of the site is loaded into the JavaScript interpreter engine of the browser, and TracingSafari stores the loaded JavaScript source code in a file. Using this facility, we obtain the `LOADING` data set by automatically visiting each site via TracingSafari. However, for the `INTERACTION` data set, we visit each site manually and give meaningful mouse events manually.

In addition, we also use TracingSafari to identify the dynamically instrumented code. Richards *et al.* [17, Section 6] report 8 mechanisms to enable dynamic code instrumentation in JavaScript, and TracingSafari stores the dynamically instrumented code by the 8 mechanisms into a separate file. Even though the 8 mechanisms do not cover the complete set of all the dynamic code instrumentation mechanisms in JavaScript, we believe that they cover most of them and we take advantage of their results to identify dynamically instrumented `with` statements.

Also, we modify the JavaScript interpreter in TracingSafari for more precise counting of executed `with` statements. We describe

[4] `www.alexa.com`

[5] `www.webkit.org`

the issue related to counting `with` statements in more detail in Section 3.2.

For further analysis of the stored source code data, we have developed a static analyzer using the lexer and the parser of Rhino[6], a Java implementation of JavaScript. Our static analyzer extracts the `with` statements from the entire source data with their source locations while counting the parsed `with` statements. Using the results, we investigate the usage patterns of `with` and count the static occurrences of it.

3.2 Usage of the `with` Statement

Now we present the amount of the `with` statement used in real world. When we try to answer the questions such as "how many sites are using the `with` statement?," one subtle problem we first encounter is how we define the "used" `with` statement. In other words, the problem is which occurrences of the `with` statement we consider as used ones in a collected code corpus. One easy and intuitive solution is to count all the `with` constructs detected by the JavaScript parser. This counting method captures all the static occurrences of the `with` statement, which is very useful for any static analyzers. Hence, we first count the static occurrences of the `with` statement in a collected JavaScript code.

However, a naïve use of static counting alone is not sufficient to show the real-world usage of the `with` statement. First, it does not take any dynamically instrumented `with` statements into account. Since JavaScript allows dynamic code instrumentation and execution by the `eval` or `eval`-like functions, any `with` statement in a string is likely to be transformed to a valid JavaScript code, but the static counting method by the parser cannot catch such `with` statements occurring in a string. Moreover, it does not show the real amount of executed `with` statements. For example, the static method counts a `with` statement in a loop as just one even though it may be executed several times in real execution and also, the method counts a `with` statement in a function body even when the function is never called. This is not a "correct" counting because we consider only the `with` statements occurring in a called, therefore evaluated, function as "used" ones.

To obtain the exact number of the executed `with` statement, we modified the JavaScript interpreter of TracingSafari as described in Section 3.1 so that we can count all the `with` statements actually evaluated by the interpreter. Unlike the naïve static counting method, this execution counting method counts a `with` statement in a loop as many as the number of loop iterations. For `with` statements in a function body, the execution counting method counts them only when the function is actually evaluated.

In addition to the static counts and the executed counts, we also count the number of the dynamically instrumented and executed `with` statements for further analysis of their usage patterns. We count such dynamic `with` statements among all the executed ones by filtering out all the static ones identified by the first static counting method and counting the remaining ones.

To sum up, we identify three kinds of the `with` statement in each data set: statically loaded ones, actually executed ones by the interpreter, and dynamically instrumented and executed ones. For brevity, we call the three counting methods `static`, `executed`, and `dynamic`, and the `with` statements counted by each method static, executed, and dynamic `with` statements, respectively.

Tables 2 and 3 show the usage of the `with` statement in two data sets. In the `LOADING` data set, 15 sites among the 98 sites have 54 static `with` statements, and 36 `with` statements are executed at initial page loading time in 8 sites. The dynamic ones are approximately one third of the executed ones. In the `INTERACTION` set, the numbers of both the web sites and `with` statements significantly increase: 38 sites have 2,245 static `with` statements and 1,232 `with` statements are executed in 27 sites. Among the executed ones, the dynamic ones are 308 from 9 sites. The rightmost column of each table shows the number of unique `with` statements. We count the unique `with` statements for each site by removing all the duplicated ones in the site and counting the remaining ones; we treat a `with` statement as a duplicate of another if those two statements are exactly the same by string comparison and consider the same `with` statements in different sites as unique ones in each site if they do not have any duplicates in the same site. We find much more duplicates in the `INTERACTION` data than in the `LOADING` data, because while we load only one page in each site at loading time, providing interactions by clicking events may load more than one page and even the same page several times. The executed ones are more likely to have duplicates than the static ones due to the counts of the `with` statements in a loop or repeated calls of the same function. However, even if we consider multiple page loadings in the `INTERACTION` data, the significant increase in the number of executed `with` statements shows that user events cause many executions of `with` statements.

Note that in both data sets, executed ones are from only a small portion of static ones: by subtracting the number of the unique dynamic `with` statements from the number of the unique executed ones, we can see that in the `LOADING` data, 7 `with` statements among 54 unique static `with` statements are executed; likewise,

Type of `with`	Web sites	`with` counts	Unique `with`s
Static	15	54	54
Executed	8	36	12
Dynamic	2	13	5

Table 2. Usage of the `with` statement in the `LOADING` data set

Type of `with`	Web sites	`with` counts	Unique `with`s
Static	38	2,245	573
Executed	27	1,232	215
Dynamic	9	308	56

Table 3. Usage of the `with` statement in the `INTERACTION` data set

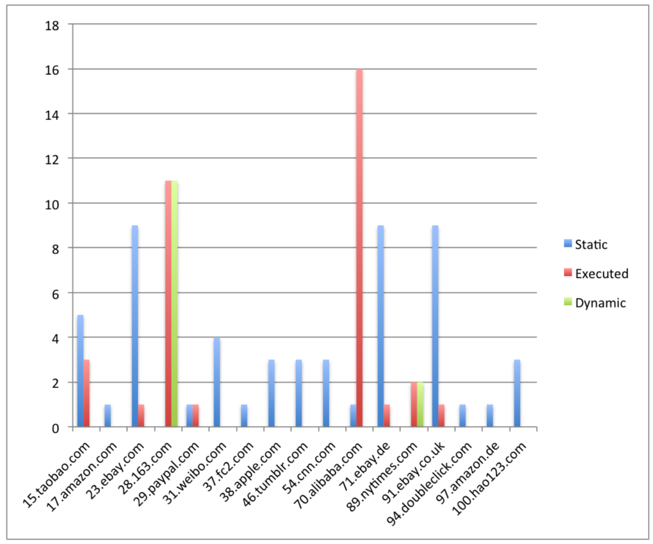

Figure 3. Number of `with` statements in each site in the `LOADING` data set

[6] `www.mozilla.org/rhino/`

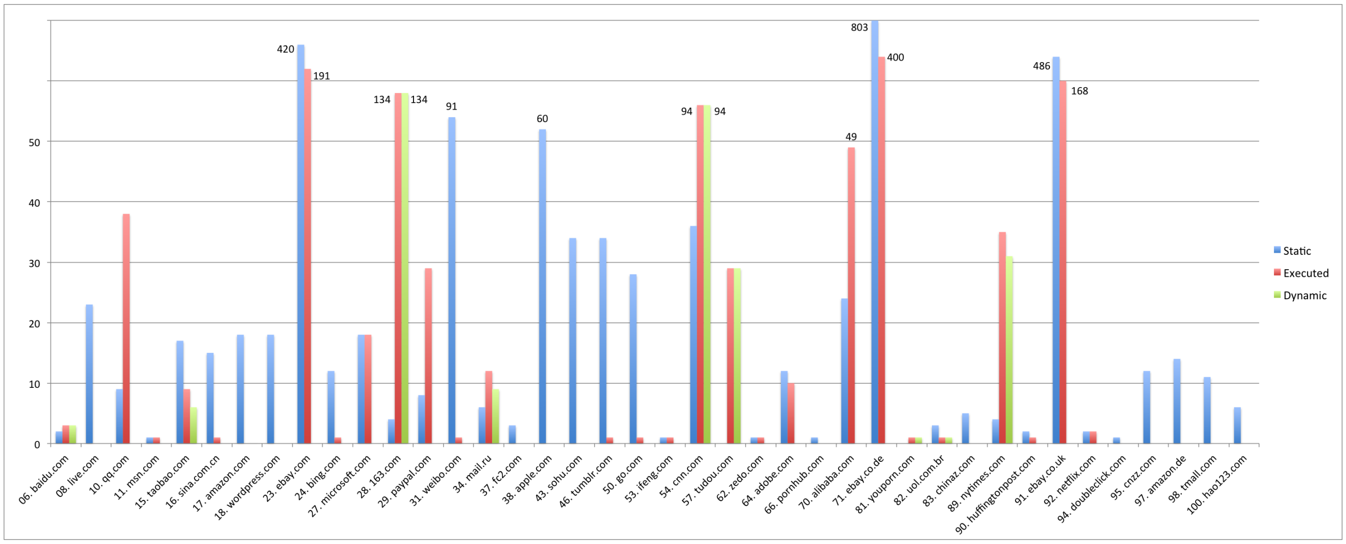

Figure 4. Number of `with` statements in each site in the `INTERACTION` data set

in the `INTERACTION` data, 159 `with` statements among 573 unique static `with` statements are executed. This means that most of the static `with` statements remain unexecuted. We think that it is because of the limited user events we provide and the coverage will be improved with more various interactions.

Figures 3 and 4 present the number of `with` statements in each site that uses `with` in the two data sets: the x-axis and y-axis indicate the web site names and the number of `with` statements in the web sites, respectively, and the prefixed number with the name is the rank of the site from the Alexa list. In the `LOADING` set, only two sites, `paypal.com` and `alibaba.com`, have all the static `with` statements executed, and most sites execute only a small portion of their static ones (4 sites) or they even do not execute the static ones at all (8 sites) at loading time. Interestingly, the graph shows that the top 10 sites do not have or use any `with` statements at loading time. The two site, `163.com` and `nytimes.com`, do not have any static `with` statements but only the dynamic ones. The `alibaba.com` site executed the most `with` statements but all the execution came from one static `with` statement. The sites affiliated with the same company showed the same statistics: 9 static and 1 executed `with` statements in `ebay.com` also appear in `ebay.de` and `ebay.co.uk`.

In the `INTERACTION` set, we found additional 23 sites which do not have any `with` statements in the `LOADING` set. Among 40 sites presented in Figure 4, 38 sites except for `tudou.com` and `youporn.com` have static `with` statements, but the static `with` in 13 sites are not executed. The top three sites that used the most `with` statements are the sites related to eBay: the `with` statements in `ebay.com`, `ebay.de`, and `ebay.co.uk` account for approximately 78% of all the static ones and 61% of all the executed ones. Interestingly, their executed `with` statements are all from the static ones. In contrast, the executed `with` statements in 5 sites, `baidu.com`, `163.com`, `cnn.com`, `youporn.com`, and `uol.com.br`, are all dynamically generated ones. Finally, the static `with` statements in `amazon.com`, `apple.com`, `doubleclick.com`, `amazon.de`, and `hao123.com` which remain unexecuted in the `LOADING` data set, also remain unexecuted even in the `INTERACTION` data set.

Our data clearly show that the `with` statement is being used unneglectably. Only with simple interactions, we observed that 38% of all the sites under consideration include the `with` statement and 27% actually executed them. We expect that more various interactions for longer time will increase the figure.

3.3 Usage Patterns

We categorize the used `with` statements into seven patterns according to the kinds of the `with` objects and how the objects are used in the body of the `with` statements. Table 4 briefly describes each usage pattern.

The DOMaccess pattern is when the `with` object is a DOM element and the body of the `with` statement accesses or changes the values of the various attributes of the DOM element or changes the structure of the current DOM. The following `with` statement from `paypal.com` shows one example in this pattern:

```
with (document.forms[k]) {
  appendChild(PAYPAL.browserscript. ... );
  appendChild(PAYPAL.browserscript. ... );
  appendChild(PAYPAL.browserscript. ... );
}
```

where `document.forms[k]` evaluates to the `k`-th DOM element in the current document and the body of `with` appends three children to the DOM element with the objects given as arguments of the `appendChild` function.

The This pattern is when the `with` object is `this`; we can access any properties of the same enclosing object without repeatedly prefixing "`this.`" within the body of the `with` statement. In addition to avoid repeated occurrences of long object accesses, this pattern has other uses as Resig and Bibeault clearly explained in their book [16]. Consider the following example:

```
function simpleCons(x) {
  var privateVar=1;
  this.publicVar=x;
  this.copyvalue=function(){
    privateVar=this.publicVar;
  }
}
var newObj = new simpleCons(3);
```

In JavaScript, any function can serve as an object constructor with the `new` keyword. The above example creates a new object, `newObj`, with the constructor, `simpleCons`. When the object is

Pattern	with object	Description
DOMaccess	DOM elements	Access or change the values of DOM element attributes
This	this	Use the same naming convention between private and public properties
Global	window	Access the global scope with the eval function
Empty	Any objects	Have the empty body
Template	Template data	Process HTML templates
Generator	Any objects	Contain dynamic code generating functions
Others	Various objects	Not categorized into the above 6 patterns but used to avoid repeatedly accessing the with object

Table 4. Brief description of usage patterns

created, the variable privateVar declared in the constructor becomes a private property of the object, and the variable publicVar and the function copyvalue prefixed with "this." become public properties. One issue with this example is that references to a private property and a public property within the constructor are different: as the body of the this.copyvalue function shows, a private property is referenced without the "this." prefix but a public property is referenced with the "this."prefix. However, using the with statement, both private and public properties can be referenced in the same way as follows:

```
this.copyvalue=function(){with(this){
  privateVar=publicVar;
}}
```

The Global pattern is when the with object is the window object and the with statement has the eval function. The window object is the global scope object in JavaScript, and it has all the global variables and functions as its properties. Hence, when the with statement introduces the window object as a new scope, the global scope takes precedence over the beforehand local scope and the execution of the body of the with statement has the same effects as the execution of it in the global environment. The following example from ebay.com shows the Global pattern:

```
with (window)
  try {
    eval(_1f);
    return true;
  }
  catch (e) {}
```

Our manual analysis found out that the variable _1f is a parameter of a function which loads a script dynamically, and the script is executed in the eval function with the global scope introduced by the with statement.

The Empty pattern is when the with statement has the empty body, usually created by a code generating function.

The Template pattern is when the with statements are used for processing HTML templates. Since templates in web documents can be replaced with any given data, they provide a way to dynamically change the contents of web pages at client side without any requests to the server for the updated pages. In the JavaScript template system, a template in an HTML document usually appears as a valid JavaScript expression with some free variables. When the system processes the template with input data usually given as a JavaScript object, the system evaluates the expression in the template using the input data, which serves as a run-time environment with bindings for the free variables in the expression to some values. Since the system gets both the template expression and input data at run time, the with statement in JavaScript makes it easy to provide the binding information simply by placing the input data object in a new current scope and the expression in the body of the with statement. An example of this pattern from 163.com is as follows:

```
with (obj) {
   _.push('<a href="', url, '">', text,'</a>');
}
```

This code generates a new <a> tag which sets the string text to the link to the web site in url. The with object obj provides the values of url and text. Hence, if the obj is the following object:

```
{url : "example.com", text : "example"}
```

then the following new tag is created at the end:

```
<a href="example.com">example</a>
```

The Generator pattern is when the body of the with statement has any dynamic code generating function such as eval, Function, setTimeout, and setInterval. According to our taxonomy of with statements, we should have put all the with statements in the Global pattern into the Generator pattern, since the ones in the Global patten have the eval function. However, to simplify the discussion of the with rewritability in Section 4.2, we separate the Global pattern from the Generator pattern: even though we cannot rewrite with statements including arbitrary dynamic code generating functions to other statements in general, we can rewrite the with statements in the Gloabl pattern to other statements as a special case as we describe in Section 4.2.

Finally, the Others pattern includes the with statements that are not categorized into any of the above patterns but the usage of them is obvious: avoiding the repeated object accesses. Consider the following example from apple.com:

```
with (this._getWindowScroll(this.options.scroll)) {
    p = [ left, top, left + width, top + height ];
}
```

This code calls the _getWindowscroll method in the enclosing object in the place of the with object and the definition of the method is as follows:

```
_getWindowScroll: function(w) {
  ...
  return { top: T, left: L, width: W, height: H };
}
```

From the return statement, we can easily see that the method returns an object which has four properties, top, left, width, and height.

Figures 5 and 6 show the number of with statements in each usage pattern of two data sets: the x-axis indicates the pattern names and the y-axis indicates the number of with statements in each pattern. In the LOADING data set, the most common pattern in the static with statements is the This pattern; all the with statements in this pattern are from three affiliated sites with eBay. The second most common pattern in static with comes from various 8 sites and it is the DOMaccess pattern; judging from the number of sites, the DOMaccess pattern is the most common pattern in static with. While the Others pattern and the Global pattern appear only in the static ones, the Template pattern appears only in the executed and dynamic ones. The Generator and Empty patterns do not appear in the whole LOADING set. The executed ones have only three patterns,

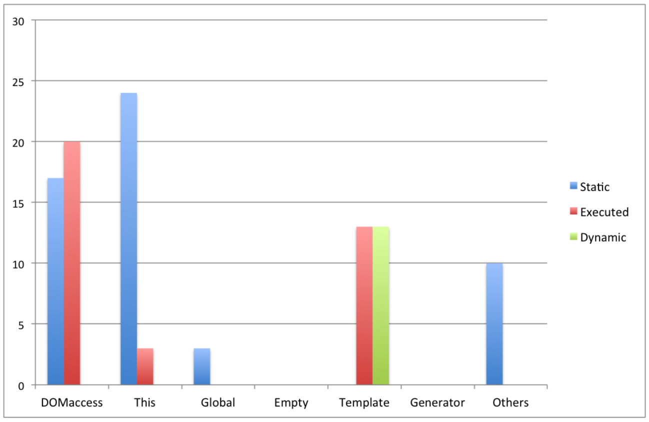

Figure 5. Number of `with` statements in each usage pattern in the `LOADING` data set

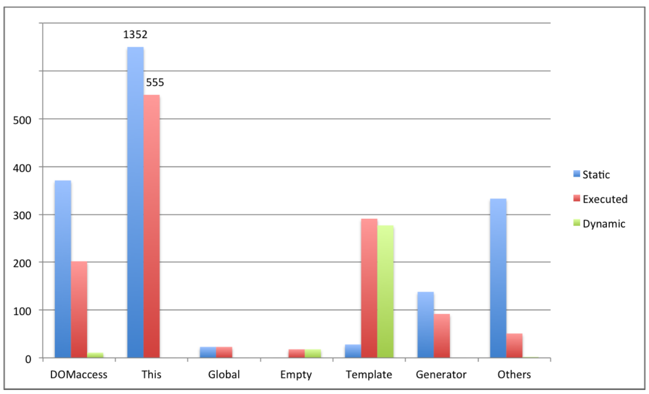

Figure 6. Number of `with` statements in each usage pattern in the `INTERACTION` data set

among which the DOMaccess pattern is the most common pattern, and the Template and This patterns follow it in order. Interestingly, all the dynamic `with` statements have only the Template pattern. In the above example from `163.com`, the `with` statement can process templates only in the `<a>` tag; in order to process templates in other tags, we need a different processing code within the `with` statement. Due to this restriction, the JavaScript template system usually generates tag-specific code dynamically depending on the input tags which contain templates. Because the `with` statements are in a string format before the code is generated, such `with` statements do not appear in the static ones.

In the `INTERACTION` set, the three most common patterns in the static `with` statements are This, DOMaccess, and Others as in the `LOADING` set. While the Empty pattern does not appear in the `INTERACTION` set either, the Generator and Template patterns additionally appear in the static ones in the `INTERACTION` set. The static `with` statements in the Template pattern do not dynamically generate code for templates, but use fixed forms for specific tags. The executed `with` statements use all seven patterns; among them, the This, Template, and DOMaccess patterns are the three most common patterns. In the dynamic `with` statements, the Template pattern is the predominant pattern in the both data sets; unlike in the `LOADING` set, those in the `INTERACTION` set additionally have the DOMaccess and Empty patterns.

Library	Homepage	`with` counts
jQuery 1.6.2	`jquery.com`	0
MooTools 1.3.2	`mootools.net`	0
Prototype 1.7	`prototypejs.org`	0
YUI Library 3.3.0	`developer.yahoo.com/yui`	0
script.aculo.us 1.9.0	`script.aculo.us`	3
Spry 1.6.1	`labs.adobe.com/technologies/spry`	0
Dojo 1.6.1	`dojotoolkit.org`	5
Ext JS 4.0.2a	`sencha.com/products/extjs`	0

Table 5. Commonly used JavaScript libraries and the numbers of `with` statements in them

3.4 The `with` Statements in Libraries

Because many web sites heavily use JavaScript libraries, analyzing the characteristics of the `with` statements in libraries helps identifying the origins of the `with` statements from libraries that are commonly used in many sites. We identified 9 major JavaScript libraries from `w3techs.com` and, except for the ASP.NET Ajax library provided as a binary format, we examined the libraries for static occurrences of `with` statements. Table 5 shows the list of the JavaScript libraries with the numbers of `with` statements in them. Only two libraries have eight static occurrences of `with` statements altogether: the script.aculo.us library has three static `with` statements, of which two have the Others pattern and one has the DOMaccess pattern, and the Dojo library has five static ones, of which four have the DOMaccess pattern and one has the Others pattern. Even though we count the `with` keywords in string formats which have a possibility to be transformed to `with` statements at run time, only one appears in script.aculo.us as an argument string of the `eval` function. This implies that the other 6 libraries do not use `with` statements as long as they do not construct the `with` statements by string manipulation and generate them at run time. Note that we do not check executed nor dynamically generated `with` statements in libraries since executing all the functionality of each library is a difficult problem and we believe that such work does not provide more useful information about the origins of the commonly used `with` statements from the libraries.

With the information of the `with` statements in libraries, we can identify the origins of some `with` statements that are commonly used in multiple sites. The `apple.com`, `tumblr.com`, and `cnn.com` sites use three static `with` statements of the same pattern, and we found out that the `with` statements are all from the script.aculo.us library. For the ones in the Dojo library, we found them only in the `INTERACTION` data set. We observed that some `with` statements have `jQuery.browser` as the `with` object, which implies that the code is from the jQuery library. However, as shown in Table 5, the latest version of jQuery does not have any `with` statements. It turned out that the sites that have the `with` statements use the jQuery 1.2.6 version, in which the jQuery team uses the same `with` statement as the ones we found, but the team replaced it with other statements without the `with` statement from the version 1.3.x.

4. Rewritability of the `with` Statement

In this section, we discuss the rewritability of the `with` statement in more detail using the real-world `with` statement examples.

4.1 Rewriting Strategy

The basic idea of the rewriting strategy is the same as what we described in Section 2.4: replace each variable occurrence in the body of the `with` statement with a conditional statement, which checks if the variable is a property of the `with` object, if so, replaces the variable access with the property access of the `with` object, otherwise, leaves it unmodified. In general, for the following `with` statement:

`with(`*exp*`)` *stmts*

we first assign *exp* to a fresh new variable as follows:

`var $f=`*exp*`;`

Then, effectively we evaluate the *exp* expression to some value before evaluating the body *stmts*. Next, we examine *stmts* in the body of the `with` statement and replace all the variable occurrences with the corresponding conditionals. if we have a variable occurrence in *stmts* as follows:

```
... id1 ...
```

we rewrite it to:

```
... ($f.hasOwnProperty('id1')? $f.id1 : id1) ...
```

In case of object property accesses such as the form of `id1.prop1`, we rewrite it to a conditional which checks only the receiver:

```
($f.hasOwnProperty('id1')? $f.id1 : id1).prop1
```

However, naïvely replacing all the variable occurrences is not sufficient especially for variable declarations and function declarations. Consider the following example:

```
with(obj) {
  var lvar1;
  alert(lvar1);
  var lvar2=3;
}
```

where two local variables, `lvar1` and `lvar2`, are declared within the `with` statement. The desired rewritten version of the code is as follows:

```
var $f=obj;
var lvar1;
alert(($f.hasOwnProperty('lvar1')? $f.lvar1
                                  : lvar1));
var lvar2;
($f.hasOwnProperty('lvar2')? $f.lvar2=3 : lvar2=3);
```

Here, if the `$f` object has a property of the name `lvar1`, the argument of the `alert` function becomes the property in the `$f` object. This might seem strange: one can think that because `lvar1` is declared in the body of the `with` statement, the local variable `lvar1` would shadow the property of the `$f` object. However, in JavaScript, all declarations are lifted up to the front of the current lexical scope and the original code is semantically the same as the following:

```
var lvar1;
var lvar2;
with(obj) {
  alert(lvar1);
  lvar2=3;
}
```

Therefore, when rewriting variable declarations, we rewrite only the initialization expressions of variable declarations not the binding occurrences of the variables. Similarly for the function declarations, even though they introduce new lexical scopes, we do not rewrite the function declarations because they are also lifted up.

The rewriting strategy for function expressions is somewhat different too. Consider the following example:

```
var fun1;
with(obj) {
   fun1=function() {
     var lvar1;
     alert(lvar1);
     alert(lvar2);
  };
}
```

The `fun1` variable is initialized to a function expression. In this case, the function is not lifted and it is created within the `with` statement introducing its own lexical scope. Although the `with` object has the `lvar1` property, the local variable, `lvar1`, in the function expression indeed shadows the property in this case, and the variable access of `lvar1` in the `alert` function becomes a local variable access. Other variable accesses not declared as local variables in the function expression are rewritten as before to corresponding conditionals. The following code shows the rewriting result of the above function expression:

```
var $f=obj;
...
function() {
   var lvar1;
   alert(lvar1);
   alert(($f.hasOwnProperty('lvar2')? $f.lvar2
                                    : lvar2));
};
```

For rewriting of function expressions, the rewriting process should maintain lexical scope information to check if variables in the function expressions reference the shadowing local variables.

For the `try-catch` statement, the same rewriting strategy applies to all the statements in the body of the `try` statement but rewriting the `catch` statement is slightly different. Assume that we have the following `catch` statement within a `with` statement:

```
catch(id){ ... }
```

In the body of the `catch` statement, while all the declarations are lifted up to the outer scope, the identifier given to the statement, `id`, is regarded as a local variable so it shadows the same named variables in outer scopes. Therefore, the rewriting process should rewrite only the variable accesses other than the accesses of `id`.

Finally, for nested `with` statements, we use nested conditionals for rewriting. For example, the following code:

```
with(obj1)
  with(obj2)
     ... id1 ...
```

where `obj2` is a variable reference, is rewritten as follows:

```
var $f1=obj1;
var $f2=($f1.hasOwnProperty('obj2')? $f1.obj2
                                   : obj2);
...
($f2.hasOwnProperty('id1')? $f2.id1 :
  ($f1.hasOwnProperty('id1')? $f1.id1 : id1))
...
```

For general cases where `obj2` is an arbitrary expression, the same rewriting strategy described in this section applies to the expression.

4.2 Rewritability of the `with` Statements in Real-World Code

Now we investigate whether the rewriting process described in the previous section is applicable to the `with` statements in each usage pattern described in Section 3. The `with` statements in the `LOADING` data set have only simple forms: they do not have function expressions nor nested `with` statements at all and most of them have only assignments, function or method calls, and variable or property accesses. In contrast, the `with` statements in the `INTERACTION` set have various forms such as all the syntactic forms described in Section 4.1 including dynamic code generating functions.

For now, we leave out the dynamically generated `with` statements and only consider the static `with` statements. From our investigation, we found that we can rewrite all static `with` statements except for the Generator pattern. We can apply the rewriting strategy directly to the `with` statements in all the rewritable patterns but in the Global pattern. To explain the issue in the Global pattern, we revisit the example of this pattern from `ebay.com`:

```
with (window)
  try {
    eval(_1f);
    return true;
  }
  catch (e) {}
```

According to the rewriting strategy in Section 4.1, the above code is rewritten to the following code:

```
var $f=window;
try {
  ($f.hasOwnProperty('eval')? $f.eval : eval)
  ($f.hasOwnProperty('_1f')? $f._1f : _1f);
   return true;
}
catch (e) {}
```

Since we know that the `eval` function in JavaScript is the property of the `window` object, the `eval` call in the rewritten version becomes the same call as the `window.eval` call. However, the calls may result in different results: while the `eval` call in the original one is executed under the global environment, the call in the rewritten one is executed under a local environment. Therefore, the rewriting strategy does not preserve the original semantics, so it is not applicable to the Global pattern.

In general, if a `with` statement includes any dynamic code generating functions such as `eval`, rewriting it statically is impossible. For example, to rewrite the following code:

```
with (obj)
   eval(str);
```

we should know what variable accesses occur in the `str` string, which is hard to figure out if `str` is constructed from complex string manipulations, thus it is not known statically. This explains why the `with` statements in the Generator pattern are not rewritable. Fortunately, however, we can rewrite the `with` statements in the Global pattern. According to the recent ECMA-262 Standard [6], we can execute the `eval` function in the global scope when calling it by aliasing. Hence, we can rewrite the above example as follows without using the `with` statement while preserving its semantics:

```
try {
  var aliaseval=eval;
  aliaseval(_1f);
  return true;
} catch (e) {}
```

where the `aliaseval` function calls the `eval` function indirectly. The same result applies to the dynamically generated `with` statements if we consider only the code that are already generated: only

Pattern	Rewritability	Rewriting
DOMaccess	Yes	Use the rewriting strategy described in Section 4.1
This	Yes	Use the rewriting strategy described in Section 4.1
Global	Yes	Remove the `with` statement and replace the `eval` function call to the alias function call of the `eval`
Empty	Yes	Use the rewriting strategy described in Section 4.1
Template	Yes	Use the rewriting strategy described in Section 4.1
Generator	No	Generally not possible due to the dynamic code generating function within the `with` statement in this pattern
Others	Yes	Use the rewriting strategy described in Section 4.1

Table 6. Summary of rewritability of the `with` statement in each usage pattern

the `with` statements in the Generator pattern are not rewritable. Consider the following revisited example from `163.com` in the Template pattern:

```
with (obj) {
   _.push('<a href="', url, '">', text,'</a>');
}
```

This code is already dynamically generated and it is obviously a simple one to rewrite.

While rewriting only the code that are already generated is too restrictive, statically rewriting any `with` which might be dynamically generated is impossible. We found that the above code is generated by the following static code:

```
B=new Function("obj",
  "var _=[];with(obj){_.push('"
  +A.replace(/[\r\t\n]/g," ").replace(...).
  ...
  +"');} return _.join(' ');");
```

In JavaScript, `Function` generates a new function object at run time which has all the string arguments except for the last one given to the `Function` as parameters and the last argument as its body. The example code generates the `with` statement through the second argument of `Function` and the second string argument is constructed by heavy string manipulations, which, for instance, divide the following tag string containing templates:

```
<a href="<#=url#>"><#=text#></a>
```

to the following four elements of two strings and two variables:

```
'<a href="', url, '">', text, '</a>'
```

As shown in Section 3.3, the two variables are replaced with some concrete values through the binding information of the argument `with` object at run time. Rewriting such static occurrences is the same as rewriting dynamic code generating functions, which is impossible in general. Therefore, the dynamically generated `with` statements are not statically rewritable.

Table 6 shows the summarized result of rewritability of the `with` statement in each usage pattern. While our rewriting strategy is not directly applicable to the `with` statements in the Global pattern, we can rewrite the static `with` statements in all patterns except for those in the Generator pattern, which contain dynamic code generating functions. This result indicates that we can eliminate most static occurrences of the `with` statement in the real-world code: we found that all static `with` statements in the `LOADING` data set and approximately 93.8% of the static `with` statements in the `INTERACTION` data set are rewritable to other statements.

5. Related Work

Richards *et al.* [17] conducted a survey on the real-world uses of the `eval` function in JavaScript applications of the 10,000 most popular web sites. They showed that, unlike what many researchers have presumed, the `eval` function is widely used and many of its uses are not for simple purposes such as JSON[7] deserialization. They also showed that some uses of the `eval` function are essential for dynamic data or code loading, but most of them could be rewritten with other features than `eval`. Their research inspired our work and a little curiosity about the actual uses of the `with` statement has been the driving force of our research. Their tool, TracingSafari, was a good infrastructure for our work to proceed.

Resig and Bibeault [16] provide a good description of the `with` statement in their book by explaining its issues in detail in one chapter of the book. Their simple experiment concerning performance overheads caused by the `with` statement inspired us to perform a similar experiment as described in Section 2. While their experiment shows some results only on an old version of one browser, our experiment shows more extensive results on four major browsers in the latest version. They also present the usage of the `with` statement occurring in some libraries and their explanation of JavaScript naming convention and template processing helped us to categorize the usage patterns of the `with` statement. However, because many sites use their own JavaScript code in addition to the libraries, the usage patterns in libraries do not represent the real usage of the `with` statement. In contrast, we show the usage of the `with` statement in the top 98 real-world web sites as well as those in the eight major libraries, and further, we discuss the issues related to static analysis and the rewriting strategy.

Now that many web pages in popular web sites contain untrusted third party JavaScript applications, isolating untrusted sources from sensitive data in a host web page has been an important issue. Major companies such as Facebook, Google, and Yahoo! have developed their own JavaScript subset languages, FBJS [5], Caja [7], and ADsafe [3], respectively, and present a solution to the isolation problem by forcing application developers to use their own subset languages. The subset languages disallow several potentially dangerous features, which include the `with` statement and dynamic code generating functions such as the `eval` function. If an application program uses such features, the static checker detects them and rejects the program. In Section 4, we showed our empirical study result that we can rewrite any static occurrences of the `with` statement in real-world applications if the `with` statement does not have any dynamic code generating functions.

As another solution of the isolation problem, Maffeis *et al.* [14] present a hybrid approach of static checks by filtering out potentially dangerous features such as `eval`, `Function`, and `construct`, and dynamic checks by rewriting and wrapping more potentially dangerous features such as `this`. Based on the operational semantics of JavaScript from their earlier work [13], they prove that their subset language guarantees the isolation property they defined: a JavaScript program does not access any designated sensitive data from the host page or another program. While their subset language includes the `with` statement, many of its use cases are restricted in the subset language though the authors did not explicitly describe what are such restrictions. On the contrary, we provide the real-world use cases of the `with` statement. It might be interesting to apply their mechanism to the real-world use cases we collected and to see whether their mechanism still applies to the real-world use cases.

The λ_{JS} [10], a small calculus defined by Guha *et al.*, captures the core part of JavaScript. Any JavaScript program without the `eval` function can be desugared into a corresponding λ_{JS} program and the authors mechanically tested the desugaring process. In addition, the authors presented a use case of λ_{JS} by defining a very simple type system for λ_{JS} which blocks accesses to a specific method, and proved the correctness of the type system. Finally, they reestablished a safe subset of JavaScript from the typed λ_{JS} system by using the compositionality of the desugraing rules. However, the subset language excludes the `with` statement and the authors add that if the `with` statement is considered important, they will extend the subset language to include the `with` statement. In Section 3, we showed that more than 30% of the top 98 sites have static occurrences of the `with` statement with some user interactions. We believe that the result shows the significance of the `with` statement.

6. Conclusion

The `with` statement in JavaScript makes static analysis of JavaScript applications difficult by introducing a new scope at run time and thus invalidating lexical scoping. To simplify the problem, many static approaches to JavaScript program analysis simply disallow the `with` statement. To decide whether to include the `with` statement, we should better understand the actual usage patterns of the `with` statement.

In this paper, we collected JavaScript code from the 98 most popular web sites and showed the real-world usage of the `with` statement: 15 sites had the total 54 static occurrences of the `with` statement in their main web page and with some simple click events, the number of web sites having the `with` statement increased to 38 with the total 573 static occurrences. This result implies that simply disallowing the `with` statement may block many use cases commonly used in some sites. In addition, we categorized the use cases of the `with` statement into the seven usage patterns. We found that the `with` statement was mainly used to keep the same naming convention inside objects between the private and public properties, or used for simple property accesses of long named objects, especially DOM element objects. The dynamically generated `with` statements were mainly used for the HTML template processing.

Moreover, we presented the rewriting strategy to desugar away the `with` statements in real-world applications and investigated the rewritability of the `with` statement in each pattern. In consequence, we found that if the `with` statement is not the dynamically generated one and does not include any dynamic code generating functions, we can rewrite all the `with` statements in real-world code to other statements without using `with`. The result is promising because all the static approaches that disallow the `with` statement also disallow dynamic code instrumentation. We are currently working on automating and formalizing the rewriting process. We believe that the work will make static analysis of JavaScript more feasible while imposing less restriction.

Finally, we plan to extend the target sites for the empirical study to the top 10,000 sites. The list of the top 98 sites shows a somewhat biased site distribution: for example, 18 sites among the 98 sites are the Google sites in different nations and three sites are the eBay sites. The web sites affiliated with the same company tend to have JavaScript code in the same pattern: while all the Google sites do not have any `with` statements, all the eBay sites have the same number of the `with` statements in the same patterns in the `LOADING` set. By extending and diversifying the target sites, we expect that

[7] JSON is a shorthand for JavaScript Object Notation, a string format for data exchange. For more information, see `www.json.org`

we can get more confidence in our results, and it will be interesting to see whether any other undiscovered usage patterns exist in the data from the top 10,000 sites.

Acknowledgments

This work is supported in part by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grant 2011-0000974).

References

[1] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *ECOOP '05: Proceedings of the 19th European conference on Object-oriented programming*, pages 429–452. Springer, 2005.

[2] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 50–62, New York, NY, USA, 2009. ACM.

[3] Douglas Crockford. ADsafe. `http://www.adsafe.org/`.

[4] Douglas Crockford. with Statement Considered Harmful. `http://yuiblog.com/blog/2006/04/11/with-statement-considered-harmful/`.

[5] Facebook. FBJS. `http://developers.facebook.com/docs/fbjs/`.

[6] European Association for Standardizing Information and Communication Systems (ECMA). ECMA-262: ECMAScript Language Specification. Fifth edition., 2009.

[7] Google. Caja. `http://code.google.com/p/google-caja/`.

[8] Salvatore Guarnieri and Benjamin Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for JavaScript code. In *SSYM '09: Proceedings of the 18th conference on USENIX security symposium*, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.

[9] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for ajax intrusion detection. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 561–570, New York, NY, USA, 2009. ACM.

[10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *ECOOP '10: Proceedings of the 24th European conference on Object-oriented programming*, pages 126–150, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] Phillip Heidegger and Peter Thiemann. Recency types for analyzing scripting languages. In *ECOOP '10: Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183, pages 200–224. Springer, 2010.

[12] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 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*, pages 270–283, New York, NY, USA, 2010. ACM.

[13] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *APLAS 2008: Programming Languages and Systems, 6th Asian Symposium, Bangalore, India, December 9-11, 2008. Proceedings*, volume 5356, pages 307–325. Springer, 2008.

[14] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *ESORICS 2009: 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, volume 5789, pages 505–522. Springer, 2009.

[15] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *S&P '10: IEEE Symposium on Security and Privacy*, pages 125–140. IEEE Computer Society, 2010.

[16] John Resig and Bear Bibeault. *Secrets of the JavaScript Ninja*. Manning Publications, 2011.

[17] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: a large-scale study of the use of eval in JavaScript applications. In *ECOOP '11: Proceedings of the 25th European conference on Object-oriented programming*. Springer LNCS, 2011.

[18] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2010. ACM.

[19] Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *ESOP 2005: Programming Languages and Systems, 14th European Symposium on Programming, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3444, pages 408–422. Springer, 2005.