2014 IEEE 38th Annual International Computers, Software and Applications Conference

JavaScript: The Used Parts

Sharath Gude and Munawar Hafiz
Auburn University
Email: munawar@auburn.edu

Allen Wirfs-Brock
Mozilla Corporation
Email: allenwb@mozilla.com

***Abstract*—We performed an empirical study to understand how different language features are used by JavaScript developers in practice, with the goal of using this information to assist future extensions of JavaScript. We inspected more than one million unique scripts (over 80 million lines of code) from various sources: JavaScript programs in the wild collected by a spider, (supposedly) better JavaScript programs collected from the top 100 URLs from the Alexa list, JavaScript programs with new language features used in Firefox Add-ons, widely used JavaScript libraries, and Node.js applications. Our corpus is larger and more diversified than those in prior studies. We also performed a study on 45 JavaScript developers to understand the reasons behind some of their language feature choices. Our study shows that there is a wide-spread confusion about newly introduced JavaScript features, a continuing misuse of existing problematic features, and a surprising lack of adoption of object-oriented features. Understanding JavaScript programming practices will assist many stakeholders: tool developers can analyze the requirements of better tools and more responsive IDEs and programmers can learn about the used (and the good) parts of JavaScript.**

Keywords*—*JavaScript, Empirical Study, Language Evolution, Language Feature.

I. Introduction

JavaScript[1] is a language that is rapidly evolving. Because it is the only general purpose programming language that is a member of the suite of standards that define the technology of the World Wide Web content, any change to the language has a potential of very broad impact. Much of the JavaScript code is passively stored on web servers and is loosely maintained. If a JavaScript feature is modified or removed in a revision of the language that becomes widely deployed by web browsers, the passively maintained web content may never be updated to remove dependencies upon deprecated JavaScript features. This means that over time, valuable legacy web content will degrade to an unusable state unless great care is taken in decisions to evolve the language. As of January 2014, the World Wide Web is estimated to consist of over 1.81 billion pages [1]. Even a minor "breaking change" to an obscure JavaScript language feature has the potential to impact thousands of web pages.

Knuth [2] pointed out more than forty years ago that designers of languages rarely have any idea how programmers use the languages; the comment still applies to JavaScript. Past attempts to include or exclude JavaScript features have been done without a scientific study of how the language features are actually used by developers. Some studies focused on JavaScript's dynamic behavior [3], [4] and its security issues [5]–[7], but these did not look at broader feature usage and adoption and did not examine a large and diverse corpus.

[1]JavaScript is the common name; ECMAScript is the name used for the language in its specification. Both names are frequently used interchangeably.

Empirical usage data is also important for browser vendors who build browsers that run the scripts, IDE vendors who build IDEs for JavaScript development, and researchers and tool builders who work on power tools for program analysis, debugging, refactoring, etc. Most importantly, the data can be used to suggest guidelines to JavaScript developers. Crockford [8] identified a few 'good' parts, 'bad' parts, and 'awful' parts of the language. His work has provided guidance, but has also generated strong opinions about the subjective judgments behind the recommendations [9]. With usage data, developers can be 'empirically persuaded' to write better scripts.

We performed an empirical study on a very large corpus of JavaScript code—*larger and more diversified* than any other existing studies. Our corpus includes scripts collected from spidered pages, top 100 Alexa pages, Firefox Add-ons pages, JavaScript libraries, and Node.js applications—over 4 million scripts in total (over 1 million unique scripts). These represent scripts written by developers with different levels of JavaScript expertise and scripts that have gone through different levels of validation process. We instrumented Mozilla Firefox to collect and analyze the scripts; for Node.js applications, we instrumented the V8 JavaScript engine. We applied a combination of static and dynamic analysis approaches to query language features. The entire process is automated and can be repeated periodically to monitor the change in usage over time.

We also performed an email-based study on developers of Node.js applications to understand why they choose some language features. The 45 participants gave us invaluable insight on how developers rationalize their choices.

We discovered several important usage facts that can help different stakeholders.

- We found confusion and misconception about recently introduced features that impact their adoption (strict mode, Section VI). This suggests that future extensions to the language will need to be carefully and broadly introduced to JavaScript web developers.
- We found continuing misuse of existing problematic features including block level declarations (Section VIII) and `for...in` iteration statements (Section X). This validates the inclusion of improved alternatives that have been proposed for the next ECMAScript revision.
- We found that adoption of non-standardized and deprecated features make it difficult to introduce new features (function in block, Section IX) and correct past mistakes (`with`, Section VII).
- We found wide use of functional programming constructs but virtually no examples of scripts defining new object abstraction (Section XI) in web applications. However, when JavaScript has been adopted in newer application

domains, as in Node.js applications, we found more adoption of object-oriented features. This shift in the usage patterns should be accounted in future extensions.

This paper makes the following contributions.

- It describes a new empirical study on a large corpus of JavaScript that includes scripts from several different sources that cover different usage perspectives of JavaScript (Section II). The focus of the study is different than other previous studies (Section V).
- It describes an automated and repeatable analysis mechanism that combines static and dynamic analysis (Section IV). This allows interested parties to perform similar studies and through periodic use to monitor the evolution of JavaScript usage patterns on the Web.
- It shows how personal survey results can be used to aid in interpreting the data obtained from automated program analysis (Section III).
- It focuses on the adoption of recently added language features and problematic features, and discusses how the empirical data can be interpreted to inform the evolution of ECMAScript (Sections VI to XI). This should guide designers of other languages.

More details about the corpus, the empirical study, the analyses, and the results are available on the project webpage: http://munawarhafiz.com/research/jssurvey/.

II. Corpus of Javascript Code

The existing studies of JavaScript focus on scripts collected from the top webpages identified by Alexa, well-known JavaScript libraries, and/or benchmark programs. Recent work on existing JavaScript benchmarks suggest that the benchmarks are not representative of real world JavaScript code [3], [10]. Alexa pages are a popular source, but even they represent a small and perhaps atypical sample of scripts. Most studies only cover the top 100 Alexa pages.

We collected scripts from various sources so that they represent JavaScript written by people with different levels of expertise and scripts that have gone through different levels of validation process (Table I). The corpus includes:

(1) Scripts collected from the wild written by developers of various skill levels.
(2) Scripts written for popular webpages that are perhaps written more carefully by experienced developers.
(3) Scripts that have gone through a validation process and contain newly introduced JavaScript features.
(4) Scripts written by experts and widely reused as libraries.
(5) Scripts written by developers who work in new application domains of JavaScript.

Scripts collected from the wild. Using a web crawler (Win Web Crawler [11]), we collected 100,000 URLs. There were many duplicates. After removing them, we wrote a script to load the remaining 66,145 URLs sequentially in an instrumented browser (Mozilla Firefox v21.0a). For each URL, the script instantiated a new browser and loaded the URL. This was done to include URLs with frame-busting [12] in the study.

The instrumented browser stored all scripts inside the loaded URL. We collected 3,621,184 scripts. Many scripts were not unique—some JavaScript libraries (e.g., jQuery) were reused. Besides, each Firefox instance loaded a few default scripts. We used the MD5 hash of a script to identify syntactically similar scripts. After removing these duplicates, there were 1,041,325 (28.76%) unique scripts. We refer to these scripts as ‘Spidered Pages’ in the rest of the paper.

TABLE I. Corpus of JavaScript Code

Script Source	Source URLs/ Programs	# of Scripts	# of Unique Scripts	% Unique	Program Size (KLOC)
Spidered Pages	66,145	3,621,184	1,041,325	28.76%	68,737
Alexa Pages	100	421,317	124,828	29.63%	10,522
Firefox Add-ons	50	1,074	991	92.27%	164
JS Libraries	3	3	3	100.00%	7
Node.js Appl.	50	2,653	2,548	96.04%	1,107
		4,046,231	1,169,695		80,537

Scripts written for popular pages. We used Win Web Crawler to automatically crawl each of the top 100 Alexa sites and collect URLs; then we loaded the URLs in the instrumented browser. Some banking websites, such as Bank of America and Chase, require a back account. We excluded these sites and considered the next top sites. In total, we extracted 421,317 scripts, out of which there were 124,828 (29.63%) unique scripts; we refer to this source as ‘Alexa Pages’.

Scripts that have been validated. We downloaded the top 50 Firefox Add-ons (25 most popular and 25 highest rated) during July 2013 and collected 991 unique scripts (using MD5 hash like before). These scripts enhance user experience (e.g., Greasemonkey), enable secure browsing (e.g., Adblock Plus), and provide power tools for developers (e.g., Firebug). Since Mozilla uses volunteers to vet each new extension and revision before posting it on their official list of approved Firefox Add-ons, these scripts had gone through some validation process. Additionally, some of these scripts contain new ECMAScript features (Mozilla Firefox actively adopts newly proposed, yet to be standardized features). We call these ‘Firefox Add-ons’.

Scripts reused as libraries. We created custom client scripts that loaded 3 popular libraries: Prototype JavaScript framework version 1.7, jQuery version 1.8.3, and YUI version 3.6.0. There were three scripts, one for each library. We refer to this source as ‘JavaScript Libraries’ in the rest of the paper.

Scripts from new application domain. We downloaded the top 50 ‘most starred’ Node.js applications as listed in www.npmjs.org during December 2013 and collected 2,548 unique scripts; we refer to this source as ‘Node.js applications’. We analyzed these scripts to explore how often developers create custom objects and use encapsulation, since scripts collected from spidered pages and Alexa pages showed a surprising lack of adoption of these object-oriented features.

III. Explanatory Study on Language Usage

In order to understand the reasons behind some of the choices made by developers, we launched an empirical study on JavaScript developers. We concentrated on Node.js application developers in this study: specifically, whether they create custom objects in their applications. We also asked a few general JavaScript usage questions.

We selected the participants from the developers of top 100 'most starred' Node.js applications as listed in www.npmjs.org. We also considered the top 100 'most prolific' developers listed in the same site. From these sources, there were 137 developers. We sent them an email with seven questions. Three questions asked about their design choices: whether they used strict mode, whether they declare variables in block scope, and whether they create custom objects in their applications. The remaining four were open-ended questions asked about the reasons behind their choices—one each for the three questions, and one about their use of mixins to extend objects. We sent the questionnaire only once during December 2013. We received 45 responses (45/137 ≈ 32.86% response rate).

IV. Mechanism of Collecting Information

We analyzed the scripts in our corpus using a combination of static and dynamic analysis. For spidered pages and Alexa pages, a custom loader loaded each URL into a Mozilla Firefox v21.0 browser with instrumented SpiderMonkey JavaScript engine. We instrumented the parser component of SpiderMonkey to analyze scripts statically while they are loaded and store the information about language features in a trace file. For example, every declared variable is analyzed by the parser component to report whether it is declared within block scope.

To analyze the dynamic features of JavaScript, e.g., whether a `with` statement is resolved in global scope, SpiderMonkey's interpreter component is instrumented; it generates a dynamic trace when a script is interpreted.

For JavaScript libraries, our client script calls and loads the target libraries in the instrumented browser, triggering the parser and interpreter components. For Node.js applications, we instrumented the V8 engine of Chrome browser and loaded the applications similarly; we studied Node.js applications only to collect information about how object-oriented features are used by developers. All the trace files were analyzed later by simple aggregator programs written in Java.

For Firefox Add-ons pages, we performed the analyses manually since the add-ons could not be loaded in isolation.

Except for the analysis of Firefox Add-ons pages, the entire process is automated and can be repeated periodically. Since the URLs are actually loaded in the instrumented browsers during the analyses, the results denote the current snapshot of the dynamic Web. We ran the analyses in January 2014. The analyses can be run periodically over target URLs to understand the evolution of scripts. All the components necessary for repeating the study is available at the project web page (https://sites.google.com/site/jsempiricalstudy/).

V. Research Questions

We identified questions relevant to three broad constituents: language designers working to develop future versions of ECMAScript; language, IDE, and tools vendors who implement and support development using JavaScript; and, developers who want guidance on how to write good JavaScript.

Specifically, we focus on the language designer's perspective. From our experience in evolving the language (author Wirfs-Brock is affiliated with the ECMAScript standards committee), we concentrate on collecting usage data of recently introduced features and features that will be modified/deprecated in the next major ECMAScript revision. We analyze usage questions concerning ECMAScript 5 strict mode, the `with` statement, block level variable declarations, functions defined in blocks, object property enumerations using the `for...in` statement, and the definition of new object abstractions.

Strict mode is a recently introduced language feature. The `with` statement is a legacy statement that is unavailable in strict mode. Block level scoping, although not currently supported, is being considered as a new feature in the next revision. Usage patterns of the `for...in` statement that enumerates over object properties provides clues about how JavaScript programmers currently make use of objects and inheritance and are relevant to the design of new enumeration mechanism in the next ECMAScript revision. Analysis of object abstractions patterns provide information about how often JavaScript programmer define new classes of objects. Sections VI to XI has the details.

VI. Use of strict mode

A. Motivation

Strict mode [13], introduced in ECMAScript 5, disallows some error-prone and hard-to-optimize features (e.g., `with`) of the language. In strict mode, JavaScript runtime generates parse-time errors and run-time exceptions rather than silent fails for some specific problematic coding patterns.

Strict mode can be enforced on a script by using the `use strict` directive. A `use strict` directive is simply an ECMScript Expression Statement that consists of the single string literal `use strict`. In the absence of strict mode support, such a statement has no observable effect upon a program.

The directive can be used in two ways: (1) declaring `use strict` at script level as a declarative prologue thus enforcing strict mode on the entire content of that file, or (2) declaring `use strict` at the beginning of a function body thus enforcing strict mode only on the function and any nested functions.

Strict mode was introduced to gradually eliminate the usage of error-prone practices and language features in future revisions of ECMAScript. But this will only succeed if the mode is widely adopted. What do the empirical data suggest?

B. Approach

We lexically searched for instances of `use strict`, written with single quote and double quotes. We used regular expressions to separate script-level and function-level declarations. We also collected information from our study participants about why they use strict mode or refrain from it.

C. Result

Table II shows how strict mode is used. Overall the adoption of strict mode is currently low, less than 1% for both spidered pages and Alexa pages. Of the 50 add-ons, only 5 use strict mode. None of the libraries use strict mode.

Most programmers declare `use strict` at function level, limiting the impact of strict mode only inside a function. But the trend is opposite in add-ons and Node.js applications.

TABLE II. STRICT MODE DECLARED IN JAVASCRIPT

Script Source	# Unique Scripts Using `use strict`	% Unique Scripts Using `use strict`	`use strict` instances	
			File Level	Fn Level
Spidered Pages	9,265	0.89%	239	27,384
Alexa Pages	965	0.77%	93	2,941
Firefox Add-ons	85	6.50%	73	12
JS Libraries	0	0.00%	0	0
Node.js Appl.	237	9.30%	166	135

D. Discussion

1) Why is strict mode unpopular?: The low adoption rate raises a concern: a future adoption of JavaScript with only safe features will be challenging.

There are three reasons why we found limited use of strict mode. First, developers cannot use new ECMAScript features (e.g., strict mode) unless they are supported by all widely used web browsers. To date, strict mode is not fully supported by all browsers. Microsoft Internet Explorer did not support strict mode until IE10 (Oct 2012), even though strict mode had been a part of ECMAScript 5 since December 2009. In these browsers, the strict mode directive will be treated as a comment. Second, there is a common misconception that strict mode changes semantics and will break the code in old or non-compliant browsers. Developers wrongly assume that using strict mode would make their code incompatible. Third, developers are not aware of the benefits that strict mode provide. They may think that the strict mode is too restrictive and similar benefits may be provided by other tools.

These reasons derive from the Node.js application developers' responses. In that community, strict mode is used more: 17 developers (17/45 ≈ 37.78%) responded that they use strict mode (Supported by Table II). But their responses to justify the choice were full of common misconceptions. Compatibility with older browsers was a concern cited by many (7, 7/45 ≈ 15.56%). For example, one responder said, "I think using strict mode for debugging, but not for production, is a reasonable approach—especially if there are concerns about how strict mode is treated on different platforms." But the concern is not true, since older browsers that do not support strict mode will only parse the directive as an unused comment.

10 developers (10/45 ≈ 22.22%) were concerned that strict mode will change the semantics of existing code or will break it ("... it is a viral mode that does not interact well with normal code."). These beliefs are mostly misconceptions. Strict mode does change some semantics, but the changes are minimal [13]. One change in semantics in strict mode is in the scoping of declarations contained within direct `eval` calls. We searched the entire corpus and did not find any instance where `eval` is used in strict mode. However, this may be because programmers removed `eval` proactively from their code when introducing strict mode—a recommended practice anyway [14]. The other concern about mixing strict and non-strict code is also a misconception. strict mode is lexically scoped to a script or function body; it has no global effect. Concatenation has problematic issues, but using separate strict and non-strict scripts within the same program is just fine.

A surprisingly high number of developers, even two of those who admit using strict mode, mentioned that strict mode is unnecessary (14, 14/45 ≈ 31.11%). 4 developers (4/45 ≈ 8.89%) mentioned that the same benefits can be found from JSLint and JSHint. A few others (4, 4/45 ≈ 8.89%) think that the mode is too restrictive, and do not support their favorite features ("Lack of octal mode is my only complaint").

2) Script level or function level?: Script-level declaration may cause a concatenation bug when a script written using strict mode is concatenated with third-party scripts that depend upon legacy features or semantics that are unavailable in strict mode. We encountered a few concatenation bugs (less than 10) while running our script on the instrumented Firefox browser to load the list of URLs. But if the developers have control over how the scripts are used (as developers of Firefox Add-ons pages and Node.js applications have, Table II), there are no problems in using the mode at script level. JavaScript libraries do not use strict mode. Library writers may believe there are issues with using strict mode but that is basically the same misunderstanding about the nature of strict mode.

Function level declaration of strict mode require more effort, but allows fine-grained control on how the mode is used.

E. Impact

Language designers should be concerned about strict mode and similar confusions about newly introduced features. A possible reason for this misconception is that JavaScript developers do not receive information from a single source. Unlike single vendor languages, there is not a coordinated 'launch' of new a JavaScript version. Instead, it rolls out at different times in different browsers sometimes in a piecemeal fashion.

Developers complained about strict mode not supporting their favorite features. Such concerns should motivate new features (e.g., new syntax for octal literals in ECMAScript 6).

VII. WITH KEYWORD IN JAVASCRIPT

A. Motivation

JavaScript's `with` statement is one of its most controversial features, denoted as an 'awful' part [8]. It was originally intended to provide a shorthand for writing recurring accesses to object properties. But it prevents lexical optimizations and complicates analysis by introducing a new dynamic scope.

Despite the controversy, do programmers use `with` in practice? If they are using `with`, how many of the instances are resolved in global scope?

B. Approach

We instrumented the interpreter component (jsinterp.cpp) of Firefox. Scripts from Firefox Add-ons pages were analyzed manually. This is because a client that exercises a library (or an add-on) may not execute all parts of the code.

C. Result

Table III shows the number of `with` statements used in various scripts. Less than 1% of unique scripts contain `with` statement. However, at runtime, many properties are resolved in global scope (11.71% in spidered pages, 10.82% in Alexa pages). Amazon and Ebay use `with` statement a lot. More than 60% of all `with` statements come from these two.

11 out of 50 Firefox Add-ons use **with**, although none of the 87 instances resolve **with** statements in global namespace. Two have the most number of **with**-s: NAMFox, an automated marker for Neoseeker, has 34 **with** statements and Feedly, a Google reader, has 25. No JavaScript libraries use **with**.

TABLE III. USAGE OF JAVASCRIPT WITH STATEMENT

Script Source	# Unique Scripts Using `with`	% Unique Scripts Using `with`	# of `with` instances	# properties resolved in global	% properties resolved in global
Spidered Pages	6,237	0.60%	8,645	1,589	11.71%
Alexa Pages	718	0.58%	1,193	719	10.82%
Firefox Add-ons	87	8.77%	87	0	0.00%
JS Libraries	0	0.00%	0	0	0.00%

D. Discussion

Most of the **with** statements are resolved within the object. The global objects on which **with** is used are predominantly objects for DOM access and window objects. For example, we counted 627 properties to be resolved in the global CSSProperties object, and 403 in window object in spidered pages.

E. Impact

ECMAScript 5 strict mode disallows **with** since it introduces dynamic scope. It is unlikely that **with** statement will ever be eliminated, because that would break existing web pages. However, the committee hopes that use of strict mode will minimize the occurrence of **with** in newly created content.

Our results also show that there are only a few instances of **with** that are resolved in global space. The remainder can be replaced with simple idioms [15]. Park et al. [6] performed an empirical study on the usage of **with** and found that they can be replaced automatically, but their study was done on a small corpus. There may be a JavaScript-specific refactoring that replaces **with** statements with suitable idioms.

VIII. VARIABLE SCOPE IN JAVASCRIPT

A. Motivation

JavaScript variables do not have block scope; they are declared with function scope. Anything declared within a code block inside a function applies throughout the function. This is confusing for programmers familiar with block scope in languages such as C and Java.

Consider the following JavaScript code.

```
function VariableScope() {
  var blockScopeOnly, functionScopeOnly, bothScopes;
  if (functionScopeOnly < 20) {
     console.log (bothScopes);
     var blockRestricted, escapedBlockScope;
     blockRestricted = blockScopeOnly;
     }
  escapedBlockScope = ...;
```

The code snippet shows JavaScript variables with various types of scopes in use. There are five usage patterns:

(1) A variable is declared in function scope and used exclusively in function scope, i.e., not used inside any blocks (`functionScopeOnly`).
(2) A variable is declared in function scope but used exclusively in block scope (`blockScopeOnly`).
(3) A variable is declared in function scope and used in both scopes (`bothScopes`).
(4) A variable is declared in block scope and used exclusively in block scope (`blockRestricted`).
(5) A variable is declared in block scope but used outside the block scope (`escapedBlockScope`).

Crockford [8] suggests that variables declared in block scope should be hoisted to the top of a function, so it is apparent that they are actually in function scope. But do people follow this advice? Are variables declared inside blocks typically referenced outside the blocks?

B. Approach

We instrumented the parser component of Mozilla Firefox (parser.cpp) to identify the scopes of variables. We automatically loaded the scripts from spidered pages, Alexa pages, and JavaScript libraries in the instrumented browser. The instrumentation keeps track of all variable's declarations and updates their scopes every time a new use of a variable is encountered; this is done per function. For scripts in Firefox Add-ons pages, we manually counted all variables and their scopes. We also asked developers about whether they declare variables inside blocks or hoist variable declarations at function level; we asked them to justify their choices.

TABLE IV. HOW DO PROGRAMMERS USE VARIABLE SCOPE?

	Declared at Function Level			Declared within Block		
Script Source	% Function Scope Only	% Block Scope Only	% Both Scopes	% Block Scope Only	% Escaped Block Scope	Total No of Variables
Spidered Pages	0.25%	0.04%	71.64%	25.68%	2.39%	24,228,649
Alexa Pages	1.03%	0.43%	69.88%	26.93%	1.73%	864,622
Firefox Add-ons	8.69%	3.66%	60.12%	27.53%	0.00%	11,918
JS Libraries	0.38%	0.22%	85.88%	13.15%	0.37%	4,574

C. Result

Table IV shows the pattern of variable declaration and usage. Of the 1,041,325 unique scripts in the corpus from spidered pages, there are over 24 million variables declared within functions. Most variables are declared in function scope and used in both scopes (71.64%). However, there are over 6 million variables (25.68%) that are declared inside a block and used within the block. Over half a million variables (2.39%) escape block scope. Variable declarations in Alexa pages share the same pattern as shown in Figure 1.

Mozilla Firefox supports the `let` construct to declare block-scoped variables in add-ons. 11 out of 50 Firefox Add-ons from our corpus use `let`. There are 1,800 variable declarations with `let`. In Table IV, they are counted as block-restricted variables.

D. Discussion

1) Similar trends in variable declaration: Figure 1 shows how variables are declared. There are similar trends in variable declarations. Most variables are declared in function scope, even if they are used in block scope. Among the variables declared in block scope, many escape in spidered pages and Alexa pages although the percentage is low. Developers of Firefox Add-ons pages and libraries, supposedly more proficient, do not allow variables to escape from block scope.

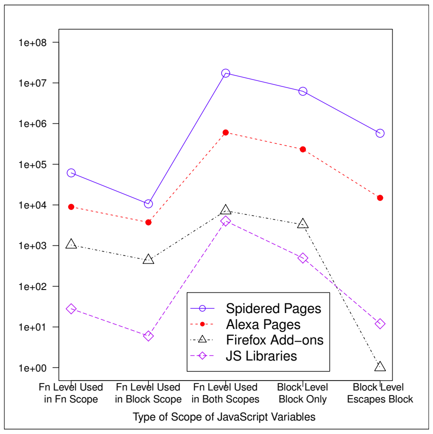

Fig. 1. Trends in Variable Declaration

2) Hoisting variable declarations: The results show that developers very loosely follow Crockford's advice and hoist block-scoped variable declarations to the top of functions. Although there are more than 10,000 instances of variables whose declarations have been hoisted to function scope in spidered pages, the proportion is very low (0.04%) compared to variables declared within blocks. Another interesting issue is that there are more variables that escape block scope compared to variables that are hoisted.

3) Do developers have a strong reason in favor of one style of declaration?: We asked 45 developers whether they have a reason behind the way they declare variables—34 replied to the question. The developers were divided: 19 developers (55.88%) declare variable at function level while 15 developers (44.12%) declare at block level. Interestingly, both camps (27, 27/34 $\approx$ 79.41%) justified that their choice produces clean and more readable code.

Developers in favor of declaring variables in function level said the approach is justified because the language itself does not have block scope ("Using function scope vars as a matter of practice keeps people from accidentally causing unexpected scope problems.", "... declaring variables in block scope would be misleading"). Some said that JSLint formed this habit. Developers were concerned that it will take a long time for `let` to be supported by browsers after ECMAScript 6 introduces it; their concerns are valid (similar issues with strict mode).

Developers in favor of declaring variables in block level argued that their choice makes code more readable ("I generally prefer to declare variables close to where they are used to maintain the conceptual flow of code", "... having a big pile of variables just below a function signature tells me very little"), even citing the principle of least privilege to justify their choice ("Principle of Least Privilege. I give variables only the smallest amount of scope needed").

E. Impact

Developers of Firefox Add-ons pages use `let` (proposed in ECMAScript 6 and supported by Mozilla Firefox), a stricter enforcement of block scope. 1,800 variables are declared with `let`. The empirical data from spidered pages and Alexa pages show that developers in general abuse function scope in scripts and allow a lot of block-level declarations to be used outside the block. However, the add-ons data show that when `let` is available, it is used in an appropriate manner. This justifies the inclusion of `let` and suggests it is likely be be widely used.

Because hoisting a variable is sometimes desired and sometimes avoided, JavaScript IDE designers can think of adding refactorings to support this activity. Hoisting a variable from block scope to function scope is an obvious refactoring. Another refactoring can redeclare a variable (previously declared with **`var`**) inside a block scope using `let`, when it can determine that the variable is used in block scope only. Yet another refactoring can push a variable from function scope to block scope and promote the use of `let`.

IX. Functions Inside Blocks

A. Motivation

JavaScript treats functions as first class objects; they can be created and modified dynamically and passed as data to other functions and objects. Additionally, it allows developers to declare functions inside other functions (inner functions). Inner functions are not present in all languages (e.g., C# added this feature in version 2.0, via anonymous delegates), but they are heavily used in JavaScript.

A function declaration is a function definition that introduced a scoped name binding for a function object. Anonymous functions can be defined anywhere, but the ECMAScript standard does not permit function declarations to syntactically occur within blocks. However, most JavaScript implementations extend the ECMAScript specification by allowing block-level function declarations. But these implementations are not consistent in the binding semantics they apply to such declarations. Some implementations treat them as declarations that are implicitly hoisted to function scope exactly like **`var`** declarations. Others treat them as conditional declarations that are available (at function scope) when the block containing them has been executed. Such differences are not significant if a block level function declaration is referenced in a 'block restricted' manner (Section VIII-A).

ECMAScript 6 plans on standardizing the semantics of inner function declarations to be block scoped. This will break existing code in which inner functions escape block scope.

B. Approach

We instrumented the parser component of Mozilla Firefox (parser.cpp) to count the inner functions. We automatically loaded the scripts from spidered pages, Alexa pages, and JavaScript libraries in the instrumented browser. When the parser encounters a function declaration, it identifies whether the function is declared inside a block; it also identifies the type of block the function is in. We did not analyze scripts from Firefox Add-ons pages since they could not be loaded in isolation and manually analyzing the scripts is tedious.

C. Results

Table V shows the results from spidered pages and Alexa pages. Most inner functions are declared outside blocks, e.g., 97.95% in spidered pages. But the actual instances of functions declared inside blocks are very high: 43,538 in spidered pages. Scripts in JavaScript libraries declare functions at the function level in all but one case. In Prototype, an anonymous function is created inside an **if** block inside `addMethods` function.

TABLE V. FUNCTIONS INSIDE BLOCK IN JAVASCRIPT

Script Source	Functions declared		Functions declared		Escape Block Scope	
	# in function scope	# in block scope	% in function scope	% in block scope	# of functions	% of functions in block scope
Spidered Pages	2,083,659	43,538	97.95%	2.05%	236	0.54%
Alexa Pages	67,115	2,411	96.53%	3.47%	37	1.53%
JS Libraries	458	1	99.78%	0.22%	0	0.00%

D. Discussion

JavaScript developers heavily use inner functions. This is not surprising as a casual examination of scripts available on the Internet reveals many such functions; also some authors encourage the use of a functional programming style [8].

There are a lot of actual instances of block level function declarations although the percentage is small. Thus, any future change of semantics for such declarations will have a significant impact upon existing web content. Libraries in our corpus, written by experienced programmers, are not impacted by the change (the single function declared inside an **if** block in Prototype library is not referenced outside the block).

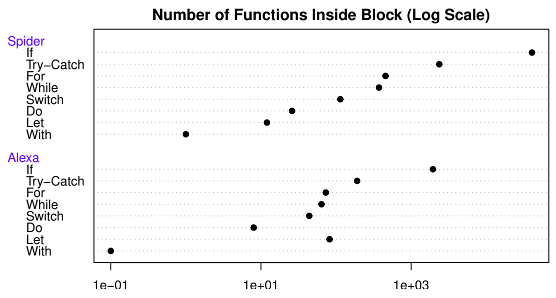

Fig. 2. Number of functions inside block

We distinguished the type of block in which the functions inside blocks are declared. Alexa pages and spidered pages show similar trends (Figure 2). Apparently, most of the functions are declared inside an **if** statement block, and **try** and **catch** statement blocks (we did not distinguish the blocks because the parser handles them together). For example, in spidered pages, among the total 43,538 functions inside blocks, most (40,612, 93.28%) are declared within an **if** block.

E. Impact

Since inner functions are used widely, they are a worthwhile focus for optimization and should be covered more in instructional materials. It also suggests that there may be opportunities for developing tools to provide refactorings that are specific to defining and using inner functions.

All browsers that support ECMAScript 5 strict mode currently disallow function declarations within blocks within strict mode code. So, lexically scoped block level function declarations can be safely added to strict mode in ECMAScript 6. The real issue is if and how such declarations can be added to ECMAScript 6 for non-strict code.

The critical factor here is the frequency of occurrence of inner function declarations that 'escape block scope'. If such occurrences are rare, interpreting existing block-level function declarations as block-scoped should not have a significant impact. Recently, Terlson [16] reported that 1.85% of survey sites are potentially broken by function in block declarations. We also calculated similar trends—37 functions (37/2,411 ≈ 1.53%) in Alexa pages, and 236 functions (236/43,538 ≈ 0.54%) in spidered pages escaped block scope.

The type of control structure that contains an inner function declaration may impact. Since functions are not currently block scoped (function bindings are hoisted to the function level), the same function object (closure) is used no matter how many times a block is entered (during a particular out function invocation). However, block scoped declaration will create a new closure each time the block is entered. This is not a significant difference for blocks that are only entered once, such as an **if** statement. But if the block is a part of a looping construct, block scoping will mean that each iteration will use a different closure that captures different outer bindings.

X. ENUMERATION WITH FOR...IN

A. Motivation

for...**in** is used for iterating through the properties of generic objects. **for**...**in** loops not only enumerate every non-shadowed, user-defined properties in current object but also enumerate the properties inherited from objects in the prototype chain. The `hasOwnProperty` construct distinguishes an object's own and inherited properties. Own properties are often used to represent per instance data values while inherited properties are typically used to represent behavioral "methods" that are shared by all instances. Thus a **for**...**in** with a `hasOwnProperty` filter is probably an indication that the code is only interested in the per instance state of an object.

Our study explores whether `hasOwnProperty` is used with **for**...**in**. Experts also suggest that **for**...**in** should not be used to iterate through an `Array`, because the standard does not specify the ordering of index elements. But do people follow?

B. Approach

We instrumented the parser component of Mozilla Firefox (parser.cpp) to identify the **for**...**in** statements and corresponding `hasOwnProperty` uses. For every **for**...**in** statement, we trace whether it is used with `hasOwnProperty`. Finding whether a **for**...**in** is used for array enumeration is similar except it is handled by a separate portion of the code inside the parser component. For scripts in Firefox Add-ons pages, we manually counted usage numbers.

TABLE VI. HOW IS FOR...IN USED?

Script Source	# of unique script with for…in	% of unique script with for…in	# of for…in instances	% of for…in using hasOwnProperty	% of for…in used for Array enumeration
Spidered Pages	72,341	6.95%	126,928	15.48%	1.84%
Alexa Pages	6,013	4.82%	11,344	36.48%	2.32%
Firefox Add-ons	295	29.77%	384	6.65%	0.92%
JS Libraries	3	100.00%	92	44.57%	0.00%

C. Result

Table VI shows the usage statistics of `for...in` alongside `hasOwnProperty`. Of the 1,041,325 unique scripts from spidered pages, there are 126,928 instances of `for...in`, but only 19,645 are used with `hasOwnProperty` (15.48%). Alexa pages show slightly better trend (`hasOwnProperty` used in 36.48% instances). Firefox Add-ons pages almost never use `hasOwnProperty` with `for...in` (only 6.65%). JavaScript libraries showcase the most appropriate use of `for...in`, with `hasOwnProperty` used in 44.57% of the cases.

Table VI also aggregates the instances of `for...in` that are used to enumerate through arrays. Out of 126,928 instances of `for...in` in spidered pages, 2,332 (1.84%) enumerate through array objects. Alexa pages show similar trend (2.32%). Library developers do not use `for...in` this way.

D. Discussion

1) `for...in` is not paired with `hasOwnProperty`: People do not follow Crockford in this case. One reason for this is that most of the objects in spidered pages and Alexa pages are stand-alone with only one parent (`Object.prototype`). Since `Object.prototype` does not have any enumerable properties, it is not necessary to use `hasOwnProperty`.

Another interesting conclusion that can be drawn is that web programmers do not favor inheritance: most of the objects are stand-alone. However, JavaScript libraries have a higher percentage of `for...in` instances with `hasOwnProperty`; they demonstrate more object-oriented feature. Prototype explicitly uses `hasOwnProperty` most of the times `for...in` is used (11 out of 18); it follows object-oriented style.

The expected pattern is to define all methods as properties of a prototype object and all data state as own properties of instance objects. A typical use of `for...in` only wants to see data properties. But prior to ECMAScript 5, there was no way for a program to define method properties as non-enumerable. This is why experts recommended `hasOwnProperty`. ECMAScript 5 introduces the `defineProperty` function that enable defining non-enumerable properties, but no uses of `defineProperty` were found in the corpus. This is probably because of the relatively recent availability of ECMAScript 5 level JavaScript implementations.

2) `for...in` used for iterating through arrays: `for...in` should not be used to enumerate through array objects. This is because the intent of array enumeration is to iterate through the array elements, but doing it with `for...in` will also iterate through named array properties. Although percentage of scripts using `for...in` in this way is generally low for spidered pages, Alexa pages show a worse trend (263 actual instances). Perhaps all those arrays do not have user-defined prototypes. But we have not tested for that.

E. Impact

The high percentage of `for...in` loops that do not contain `hasOwnProperty` may indicate that it is uncommon for developers to define their object abstractions with shared prototype methods. Instead they may be primarily using the built-in object abstractions whose methods are not enumerated by `for...in`. In section XI, we examine other indications that the definition of new object-based abstraction are rare.

A `for...of` statement has been proposed for inclusion in the next revision of the ECMAScript standard. It is similar to `for...in`, but it uses an extensible iterator abstraction for choosing the elements and their ordering for iterations. The default iterator for `Array` object will produce the array elements in index order and will not produce any non-element properties (own or inherited). The data confirming the wide use of `for...in` to iterate over arrays support the need for a feature comparable to `for...of`. It also suggests that there is probably a need for tools to support refactoring of `for...in` statements into `for...of` statements.

XI. Objects in Javascript

A. Motivation

JavaScript supports both functional and object-oriented programming style. Developers frequently follow functional style and create inner functions (Section IX). But do developers follow object-oriented style? Which objects do they create?

JavaScript developers can create new objects using: (1) object literals, (2) the `new` keyword, and (3) the `Object.create` function (only in ECMAScript 5). We explored the last two approaches that are typically used for creating complex objects.

B. Approach

We instrumented the interpreter component of Mozilla Firefox (jsinterp.cpp) to count the constructors invoked by the `new` keyword. We distinguished the objects created by the constructors. For Node.js applications, we instrumented the interpreter component of V8 (runtime.cc). We searched the scripts for `Object.create`. We also surveyed 45 Node.js developers about whether they create custom objects in their code and whether thy use mixins to extend objects; 34 developers responded.

C. Results

We randomly explored 12,000 spidered pages and 4,000 Alexa pages. We identified 5,182 custom objects in spidered pages, and 627 custom objects in Alexa pages. Many source URLs do not contain any new objects. Also, many of these are empty constructors that are then used to augment Object. But Node.js application developers create more custom objects in proportion. In the 50 applications in our corpus, we identified 71 custom objects created using `new`.

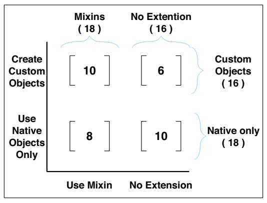

Fig. 3. Developers of Node.js applications on custom object creation

However, many Node.js application developers (18, 18/34 ≈ 52.94%) said that they use native objects only. A popular alternative that these developers may adopt is to use mixins to extend native objects. We found that half of the developers (18, 18/34 ≈ 52.94%) use mixins (Figure 3). However, about one-third of the developers (10, 10/34 ≈ 29.41&) neither define custom objects, nor extend and customize native objects.

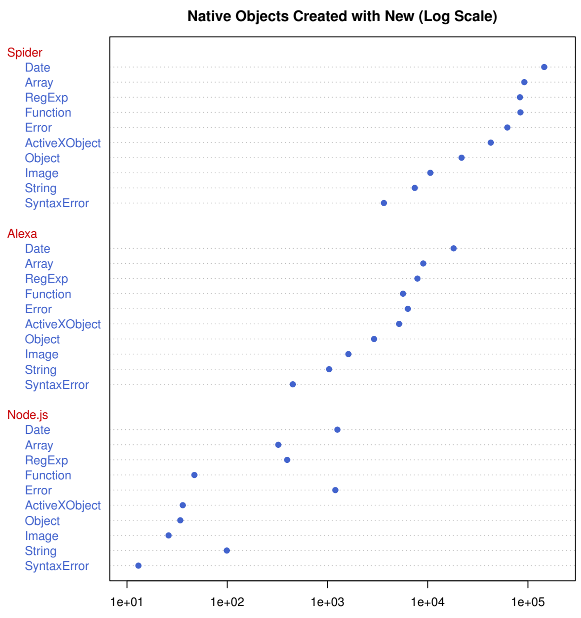

Fig. 4. Native objects created with **new**

D. Discussion

1) Constructor functions are seldom defined: The results show that developers seldom write constructor functions. This raises another question regarding how objects are created in JavaScript. We searched for the **new** keyword and what types of objects are created. Figure 4 shows the results for spidered pages, Alexa pages, and Node.js applications. They show similar trends. For example, Alexa pages contain over 60,000 instances of **new** to create objects, but almost all of them create native objects, such as `Date`, `RegExp`, `Array`, etc.

Objects can also be created by `Object.create` function. We searched for instances of `Object.create` in the scripts in our corpus. We found 4,791 occurrences in spidered pages, 539 occurrences in Alexa pages, and 165 instances in Node.js applications; on the contrary, we found thousands of instances of **new** in our corpus (Figure 4). `Object.create` instances are less than 1% of all object creation instances with `Object .create` and **new**.

2) Custom methods to customize objects: Yet another method of getting new objects is to extend existing objects using mixins. We asked the developers in our study about the method they adopt to create mixins. Developers in our survey who use mixins did not agree upon a common mechanism ("extend, DIY, third party libraries"). One developer precisely summarized the sentiment, "I'd really like to use something like traits but haven't found a great style yet. I do use mixins but only at the app level—at the library level I'm more conservative and fear collisions ... I suspect a programming style could be evolved to accommodate (or better, we'll get some syntactic support eventually)."

E. Impact

The results in this section along with the **for**...**in** results in section X suggests that the actual definition of new object abstraction is rare in scripts. This is very surprising given the amount of attention given by works such as Crockford's [8] to techniques for defining such abstractions and the effort being made to incorporate class definition syntax into in ECMAScript 6 [17].

ECMAScript 6, as currently proposed, includes both syntactic support for object abstraction (class definitions and enhanced object literals) and standard library support for mixins (`Object.assign`). Our results do not directly support the need for those features. However, current usage pattern is only one of many inputs into the language design process. It may be the Node.js applications that represent the new brand of JavaScript and indicate a latent need. Our developer survey suggests that too many competing library based solutions may be a barrier to adoption. It may be the case standardization will remove that barrier leading to wider use of application specific object abstractions by JavaScript programmers.

XII. Related Work

There have been several recent empirical studies on the dynamic properties of JavaScript. Ratanaworabhan and colleagues [3] first explored the dynamic properties of JavaScript, but their focus was to establish that the existing JavaScript benchmarks are unable to represent JavaScript code that is written. Richards and colleagues [4] had similar focus. They studied execution traces collected from 100 top Alexa pages and three industry benchmark suites to characterize the dynamic behavior of scripts and compare the dynamism with assumptions made by the benchmarks. Martinsen and colleagues [18] also found that interactive web applications (Facebook, Twitter, and MySpace) host scripts that have different execution behavior than those in benchmarks. Recent studies focus usage patterns in JavaScript programs [19], [20].

Our study focuses on how the language features are used and abused by programmers. Recently, Microsoft's Brian Terlson [16] studied top 10,000 Alexa pages to understand how people use strict mode, `const`, etc. The scope of our study and the corresponding research questions are different from his study. Even though our corpus is larger and more diversified, we chose not to repeat research questions already asked, e.g., how people use `eval` [5].

Researchers have been working on improving the way people write JavaScript code by exploring how to analyze JavaScript, and how to transform scripts to make them secure as well as elegant; many of these efforts are backed up by empirical studies to understand the JavaScript features in question. Richards and colleagues extended their work on understanding dynamism in JavaScript by studying the use of `eval` [5] and its security implications on a larger corpus (loading random pages from top 10,000 Alexa pages). This supported Meawad and colleague's [14] work on semi-automatically removing `eval` from scripts. Park and colleagues [6] studied top 98 Alexa pages and 9 JavaScript libraries to find whether **with** statements in the scripts can be replaced with JavaScript idioms. Mirghashemi et al. [21] explored a program transformation approach to rename anonymous function studying 10 JavaScript libraries. Yue and colleagues [7] studied scripts from the top 500 Alexa pages in 15 categories (removing overlaps, 6,805 pages in total) to understand how insecure practices such as `eval` and `innerHTML` are used. They suggest that safe alternatives exist for both insecure JavaScript generation and insecure

JavaScript dynamic inclusion. Nikiforakis et al. [22] studied top 10,000 Alexa pages to understand and identify the best practices of including JavaScript. Our study has similar goals. However, because of its diversity and focus on language features, it can support broader constituents—primarily language designers, but also researchers and JavaScript developers.

XIII. Threats to Validity

There are several threats to validity of our study; here we discuss how these threats have been mitigated.

The main threat is the generalizability of the results. After all, our corpus of URLs from spidered pages and Alexa pages are perhaps too small compared to the actual World Wide Web. However, our corpus is larger and more diversified than any other previous empirical studies (Section II). Using spidered pages allowed us to explore a broader range of scripts than those that are found in Alexa pages (the main focus of other studies). Firefox Add-ons pages, JavaScript libraries, and Node.js applications add diversity. The data collected from various sources also tend to validate each other.

There are specific surprising conclusions that may raise a concern. For example, we found that custom objects are rarely created by the vast majority of developers who write scripts embedded in web URLs. The result intuitively makes sense since the traditional use of JavaScript on the web was to do simple manipulations of the browser-provided DOM. However, it may happen that the emerging JavaScript application domains that are most likely to need to define new object abstractions—namely, AJAX style applications that use JavaScript to manipulate complex view models or client-side web applications that need an internal domain object model—are not properly represented in our corpus. To understand object usage in such domains, we studied Node.js applications and also surveyed developers of these applications.

Another concern is that we only studied developers of Node.js applications in our survey; perhaps that community has some idiosyncrasies. We wanted to study the developers primarily to understand the viewpoints behind object creations—developers of Node.js applications fit into that criteria. We selected the participants randomly, from a combination of most prolific Node.js developers and developers of top Node.js applications maintained in a list prepared by a popular site for Node.js applications. Their perspectives provide expert developers' insight on object creation. We also asked about their perspectives on strict mode and object enumeration with `for...in`. But these are general concerns for all JavaScript developers; and the Node.js community do not have any specific idioms or competing language features for these. The collected data is coded by the first two authors and is only reported when the two authors agreed upon the codes.

Our study provides a mechanism to take snapshots of the web at a particular time. But web applications are constantly evolving. It will need a family of experiments to understand some of the emerging trends. To achieve this, we wanted to develop a research methodology that can be repeated periodically. Most of the methodology is fully automated and repeatable. Future studies may benefit from this.

XIV. Conclusion

Evolving a language that is widely adopted yet loosely maintained is hard. It is equally hard to build support for developers: support in terms of browsers with language features, IDEs with refactorings, and materials with guidance. Our study provides empirical evidence on several important issues that will assist in understanding how to evolve the language in a way beneficial for multiple stakeholders.

Acknowledgement

We thank Mozilla Research for supporting this work and its conference presentation. We also thank the reviewers.

References

[1] M. de Kunder, "The size of the World Wide Web (The Internet)," Jan. 2014, http://worldwidewebsize.com.

[2] D. E. Knuth, "An empirical study of FORTRAN programs," *Software-Practice and Experience*, vol. 1, pp. 105–133, 1971.

[3] P. Ratanaworabhan, B. Livshits, and B. Zorn, "JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications," in *Proceedings of the 2010 USENIX conference on Web application development*. Berkeley, CA, USA: USENIX Association, 2010.

[4] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *PLDI 2010*. ACM, 2010.

[5] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do - A large-scale study of the use of eval in JavaScript applications," in *ECOOP 2011*. Springer, 2011, pp. 52–78.

[6] C. Park, H. Lee, and S. Ryu, "An empirical study on the rewritability of the with statement in JavaScript," in *FOOL 2011*, Oct. 2011.

[7] C. Yue and H. Wang, "Characterizing insecure JavaScript practices on the web," in *WWW 2009*. ACM, 2009, pp. 961–970.

[8] D. Crockford, *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[9] A. Kovalyov, "Why I forked JSLint to JSHint," Feb. 2011, http://anton.kovalyov.net/2011/02/20/why-i-forked-jslint-to-jshint/.

[10] G. Richards, A. Gal, B. Eich, and J. Vitek, "Automated construction of JavaScript benchmarks," in *OOPSLA 2011*. New York, NY, USA: ACM, 2011, pp. 677–694.

[11] "Win web crawler v2.0," http://www.winwebcrawler.com/index.htm.

[12] Grizzly WebMaster, "Javascripts frames buster," http://grizzlyweb.com/webmaster/javascripts/framesbuster.asp.

[13] Standard ECMA-262, "The strict mode of ECMAScript," http://www.wirfs-brock.com/allen/draft-ES5.1/#sec-C.

[14] F. Meawad, G. Richards, F. Morandat, and J. Vitek, "Eval begone!: semi-automated removal of eval from JavaScript programs," in *OOPSLA 2012*. New York, NY, USA: ACM, 2012, pp. 607–620.

[15] D. Crockford, "with Statement Considered Harmful," 2006, http://www.yuiblog.com/blog/2006/04/11/with-statement-considered-harmful/.

[16] B. Terlson, "Real World JS Code," Jan. 2013, http://wiki.ecmascript.org/lib/exe/fetch.php?id=meetings%3Ameeting_jan_29_2013&cache=cache&media=meetings:real-world-js-code.pdf.

[17] N. Zakas, "Does JavaScript need classes?" http://www.nczonline.net/blog/2012/10/16/does-javascript-need-classes/.

[18] J. K. Martinsen and H. Grahn, "A methodology for evaluating JavaScript execution behavior in interactive web applications," in *AICCSA 2011*. IEEE, 2011, pp. 241–248.

[19] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, "Understanding JavaScript event-based interactions," May 2014.

[20] H. V. Nguyen, H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Mining interprocedural, data-oriented usage patterns in JavaScript web applications," May 2014, to appear at ICSE 2014.

[21] S. Mirghasemi, J. J. Barton, and C. Petitpierre, "Naming anonymous JavaScript functions," in *OOPSLA 2011*. New York, NY, USA: ACM, 2011, pp. 277–288.

[22] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. V. Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: large-scale evaluation of remote JavaScript inclusions," in *CCS 2012*. ACM, 2012, pp. 736–747.