SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2016; **46**:867–889
Published online 29 May 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2334

Empirical study of the dynamic behavior of JavaScript objects

Shiyi Wei[1,*,†], Franceska Xhakaj[2] and Barbara G. Ryder[1]

[1]*Department of Computer Science, Virginia Tech, Blacksburg, VA, USA*
[2]*Department of Computer Science, Lafayette College, Easton, PA, USA*

SUMMARY

Despite the popularity of JavaScript for client-side web applications, there is a lack of effective software tools supporting JavaScript development and testing. The dynamic characteristics of JavaScript pose software engineering challenges such as program understanding and security. One important feature of JavaScript is that its objects support flexible mechanisms such as property changes at runtime and prototype-based inheritance, making it difficult to reason about object behavior. We have performed an empirical study on real JavaScript applications to understand the dynamic behavior of JavaScript objects. We present metrics to measure behavior of JavaScript objects during execution (e.g., operations associated with an object, object size, and property type changes). We also investigated the behavioral patterns of observed objects to understand the coding or user interaction practices in JavaScript software. Copyright © 2015 John Wiley & Sons, Ltd.

Received 7 April 2014; Revised 6 May 2015; Accepted 8 May 2015

KEY WORDS: JavaScript; study of websites; object behavioral metrics and patterns

1. INTRODUCTION

JavaScript is a dynamic programming language known for its flexibility of programming, supporting mechanisms for runtime code loading and generation. JavaScript is the most widely used programming language for developing interactive, client-side web applications. Based on recent usage statistics, 89% of all website software uses JavaScript [1], and there is a trend that JavaScript is becoming the most popular programming language, overall [2]. In addition, developers commonly use large libraries and frameworks (e.g., jQuery) to build JavaScript applications.

Unlike other popular object-oriented languages (e.g, Java, C, and C#), JavaScript does not have classes; instead, it supports prototype-based inheritance [3, 4]. We will discuss prototype-based inheritance in Sections 2.1 and 4.3. Furthermore, JavaScript object properties may change at any program point (e.g., property deletions). These features make it difficult to reason about the behavior of JavaScript objects. Therefore, these features pose several software engineering challenges for JavaScript applications: (i) Powerful integrated development environments (IDEs) enable more effective software development for programming languages such as Java and C#; however, there is a lack of IDE support for developing or understanding JavaScript programs. Because the properties and inheritance of JavaScript objects may change during execution, code completion suggestions are too imprecise to be useful [5]. (ii) Because JavaScript websites are exposed to web attackers, it is important to have software tools that can detect security vulnerabilities. Detecting possible security exploits becomes challenging when the approximation of program behavior is inaccurate. (iii) Optimization of JavaScript programs is difficult because of the dynamic nature of objects (e.g., dynamic typing).

*Correspondence to: Shiyi Wei, Department of Computer Science, Virginia Tech, Blacksburg, VA, USA.
†E-mail: wei@cs.vt.edu

Researchers have presented several approaches for handling such challenges in program understanding [5–7], security [8–10], and optimization [11, 12]. This work usually makes assumptions about the behavior of JavaScript programs, focusing on a subset of the language characteristics. We believe that an in-depth investigation of JavaScript applications can help researchers better understand program behavior and be more informed in choosing their techniques when designing JavaScript tools. Richards *et al.* presented an innovative study on the dynamic behavior of JavaScript applications [13]. The authors evaluated several dynamic metrics (e.g., call site dynamism, function variadicity, and uses of *eval*) of JavaScript websites (Section 2.2).

In this paper, we conduct an empirical study focusing on understanding the runtime behavior of JavaScript objects. Because of the usage of delegation in prototype-based programming, the properties and inheritance of a JavaScript object may change at runtime. This design feature makes it difficult to predict JavaScript object behavior and requires an in-depth study to better understand the runtime behavior of JavaScript objects in websites. Our study was performed on the same set of dynamic *traces* of JavaScript websites as in Richards *et al.* [13] to augment the existing observations with object behavior characteristics for these executions. Each trace consists of information from an execution such as executed code and instructions (see details in Section 3.2). Our study provides the *first* in-depth investigation of JavaScript object behavior. We report the number and kind of operations associated with objects, type changes of object properties, dynamic characteristics of prototype-based inheritance, and so on. We also summarize the *behavioral patterns* of certain objects (i.e., *user objects*) suggesting common practices that pose difficulties for program understanding. We designed specific metrics for measuring JavaScript objects and their behavior, aggregating the results via offline analyses. We relate our findings to assumptions of existing approaches for analyzing JavaScript applications.

The major contributions of this study are as follows:

- We designed metrics for measuring JavaScript objects and empirically collected and summarized results on JavaScript website traces. We measured object features including object size, dynamic typing, and prototyping. The most interesting findings include: (i) the local *size* (i.e., number of local properties) of a user object changes significantly at different program points; (ii) singleton constructors (i.e., a constructor function that only creates one observed object instance) exist widely in JavaScript applications; and (iii) prototype-based inheritance is not often implemented for code reuse.
- We investigated behavioral patterns of *user* objects and linked them to coding or user interaction practices. We studied patterns of operation occurrence sequences per object/property and patterns of property-type change. We reported several interesting JavaScript object behaviors including the following: (i) cases where read operations of the same property trigger different lookup mechanisms (local or inherited property) at different program points, making it difficult to understand JavaScript property accesses; (ii) cases where JavaScript programmers seem to intentionally use a delete operation to ensure that a certain property does not exist at a program point, not knowing if the property existed before; and (iii) cases where a property demonstrates different behaviors at different stages of its lifetime as its type changes.

Overview. The rest of the paper is organized as follows. Section 2 provides the background on JavaScript objects and then discusses related work. Section 3 describes the design of our empirical study, the experimental setup, and threats to validity. Section 4 presents the summarized metrics, and Section 5 discusses the behavioral patterns of JavaScript objects. Section 6 offers conclusions and future work.

2. BACKGROUND AND RELATED WORK

In this section, we introduce the JavaScript dynamic characteristics that may affect object behavior. We also present work related to our empirical study.

DOI: 10.1002/spe

2.1. Dynamic behavior of JavaScript objects

JavaScript is a dynamic object-oriented programming language. Its dynamic characteristics render the runtime behavior of JavaScript applications unpredictable. In addition to the reflective mechanisms that enable code generation and loading during execution, there are several program features in JavaScript that make it difficult to accurately model object behavior.

JavaScript is a dynamically typed language. A JavaScript variable may be bound to different types at different program points. Thus, static type checking, a technique widely applied in software optimization tools for programming languages such as Java and C, may be inaccurate. In this study, we demonstrated the frequency and patterns of type changes in JavaScript object properties in popular websites.

Object properties can be added, updated, or deleted at runtime. The possible behavior of a JavaScript object is defined by its properties. In JavaScript, an object property can be added or updated via an indirect assignment statement (e.g., $x.p = y$); it can also be deleted via a delete statement (e.g., *delete x.p*). This means that a JavaScript object may exhibit different behaviors at different times during execution. We have studied property changes and their effect on object behavior.

JavaScript supports prototype-based inheritance. Instead of class-based inheritance, a JavaScript object inherits properties from a chain of prototype objects that is defined during execution. Lacking the notion of class, it is difficult to summarize the type of a JavaScript object at a particular program point. The inherited properties of JavaScript objects cannot be predicted accurately because they are decided at runtime. In this study, we investigated the dynamic behavior of JavaScript inheritance.

Objects created by native languages. Many JavaScript applications executed with native code (e.g., C and C++) are unavailable to JavaScript tools. In this study, we report the behavior of the objects created by these native languages.

2.2. Related work

We present the works that are the most relevant to our research: (i) empirical studies of JavaScript applications; (ii) related analyses of the dynamic behavior of JavaScript objects; and (iii) dynamic metrics for other programming languages.

Dynamic studies of JavaScript applications. JavaScript features, which are used in the programs, introduce dynamic behavior that has been studied in previous research. Richards *et al.* performed experiments on JavaScript programs downloaded from popular websites running in the browser to study several aspects of dynamic behavior [13]. Popular websites were studied resulting in several conclusive observations: (i) the prototype hierarchy often changes within libraries; (ii) properties are not just added at object initialization; and (iii) property deletions are common in some websites. Recall that in our study, we reused these benchmarks (i.e., traces) and modified the analysis infrastructure in Richards *et al.* [13].[‡] We focused on more detailed observations about individual JavaScript object behavioral patterns. Richards *et al.* also presented an evaluation of the runtime code generation mechanisms in JavaScript, focusing on the *eval* construct [14]. The authors analyzed the details of the uses of *eval* and demonstrated several cases. We focused on other important programming language features (e.g., object inheritance through prototyping) that affect the dynamic behavior of JavaScript applications.

Ratanaworabhan *et al.* presented a study comparing the behavior of JavaScript benchmarks (e.g., *SunSpider* and *V8*) with real web applications [15]. The authors evaluated differences in behavior between the benchmarks and websites, concluding that the benchmarks were not representative of the behavior of real JavaScript applications. This study motivated us to conduct our experiments on JavaScript code extracted from websites.

Martinsen and Grahn performed a study on social networking web applications to understand the different behaviors between social networks and established benchmarks [16]. The authors focused

[‡]The traces and original tools [13] are available at https://www.cs.purdue.edu/sss/projects/dynjs/.

DOI: 10.1002/spe

on studying JavaScript function behavior. For example, they found a high variance in the execution times of individual functions in social networking applications and that anonymous functions were used frequently. The behavior of social networking applications was revealed to be very different from the benchmarks. In our study, we analyzed a larger set of web applications and focused on the behavior of JavaScript objects, not functions.

Ocariza Jr. *et al.* studied JavaScript errors in web applications [17, 18]. The authors categorized the observed errors at runtime and summarized their correlations to the characteristics of JavaScript websites [17]. For example, they observed that there was a medium correlation between the number of null exceptions and the average number of property deletions in the JavaScript code. In Ocariza Jr. *et al.* [18], the authors focused on the JavaScript bugs caused by the Document Object Model (DOM). We report more than the summarized results. In the future, we plan to study the correlation between JavaScript errors and object behavioral patterns.

Yue and Wang performed a dynamic study focusing on non-secure JavaScript practices on the web [19, 20]. The authors used an instrumented version of Firefox to collect trace files and evaluated non-secure practices (e.g., *eval*) by performing offline analyses. We focused on the nature of JavaScript objects and their behavior, but not specifically on security.

Analyses of the dynamic behavior of JavaScript objects. The goal of our study is to better understand JavaScript object behavior. We also discuss the effectiveness of existing or possible program analysis techniques based on empirical observations in Section 4. There are several analyses that apply specialized approaches for JavaScript objects. Wei and Ryder presented a context-sensitive points-to analysis to keep track of the changes of object properties [21]. This analysis used an approximated type of the receiver object as the calling context, and object properties were calculated using a partially flow-sensitive analysis. The algorithm was implemented in the JavaScript Blended Analysis Framework [10] and scaled to real websites. Our study results demonstrate that changes of object properties happen frequently in JavaScript applications; an analysis that models these changes will likely produce more accurate solutions.

Jensen *et al.* presented a static analysis that can precisely model JavaScript objects and their prototypes [22]. Their flow-sensitive analysis handled several features of JavaScript objects (e.g., prototype-based inheritance and property changes at runtime); however, it could not analyze real websites efficiently. Our study of dynamic characteristics of JavaScript objects in website code may help this analysis improve its performance by specializing the analysis for features appearing often in real code.

Guarnieri *et al.* presented a static taint analysis finding security vulnerabilities in JavaScript websites [9]. The analysis focused on addressing dynamic features of JavaScript including object creations and accesses through constructed property names. Other features (e.g., inheritance and runtime property changes) were modeled conservatively. Our study suggests the latter features should be handled more precisely to produce more accurate analysis solutions.

Madsen *et al.* presented a static analysis of JavaScript applications handling some of the challenges posed by native code [6]. The authors designed a *use analysis* combined with a pointer analysis to recover information about the structure of objects and to infer the missing inter-procedural flow introduced by the unavailable code. Our study shows that objects created by native code frequently occur in JavaScript websites, and therefore, the analysis [6] may be quite useful.

Several JavaScript static type systems (e.g., [23, 24]) were introduced despite the difficulty of building an accurate model (Section 2.1). However, there was no empirical evidence to show that these type systems considering JavaScript features (e.g., property changes) were scalable to real websites. The observations made in our paper may be useful for making reasonable approximations in the representation of JavaScript objects to increase scalability in these type systems.

Schäfer *et al.* presented a dynamic analysis to identify determinate (i.e., always having the same value at a given program point) variables and expressions in JavaScript programs [25]. The approach soundly inferred the determinacy facts that hold for any execution. The authors argued the results might be helpful to improve static pointer analysis. Similarly, Andreasen and Møller presented a static approach to infer and exploit determinacy information [26]. Our study shows that although object property changes happen frequently in JavaScript websites, there are many properties whose

DOI: 10.1002/spe

value never changes in the observed executions, indicating possible determinacy. Thus, determinacy analyses ([25, 26]) can be used on real JavaScript applications to establish its effectiveness in practice.

Dynamic metrics for other programming languages. Dufour *et al.* presented a study on Java programs applying dynamic metrics [27]. The authors developed several general metrics that also can be used for studying JavaScript applications. In our work, we adapted these ideas for measuring polymorphism and object size in the context of JavaScript objects.

Holkner and Harland conducted experiments on Python programs studying their dynamic behavior [28]. The authors focused on the instructions such as adds and deletes. Furr *et al.* presented a dynamic analysis and transformation of Ruby programs [29]. The runtime instrumentation gathered profiles of dynamic feature usage and these features were then replaced with statically analyzable alternatives. The usage reported shows that dynamic features are pervasive throughout the Ruby benchmark suite, especially the *eval* construct [29]. Python, Ruby, and JavaScript are all dynamic programming languages sharing certain language features. The results of our study may be compared with these results ([28, 29]) to illustrate similarities and differences among these programming languages.

Hills *et al.* presented an empirical study on PHP feature usage [30]. The authors presented several dynamic metrics similar to those in Richards *et al.* [13] and provided guidance for developing program analysis tools for PHP. In our study, we use the empirical results to better understand the behavior of JavaScript applications.

3. EXPERIMENTAL DESIGN

In this section, we discuss the choices we made when we conducted the study and the setup of the experiments. We also present the threats to validity of our study.

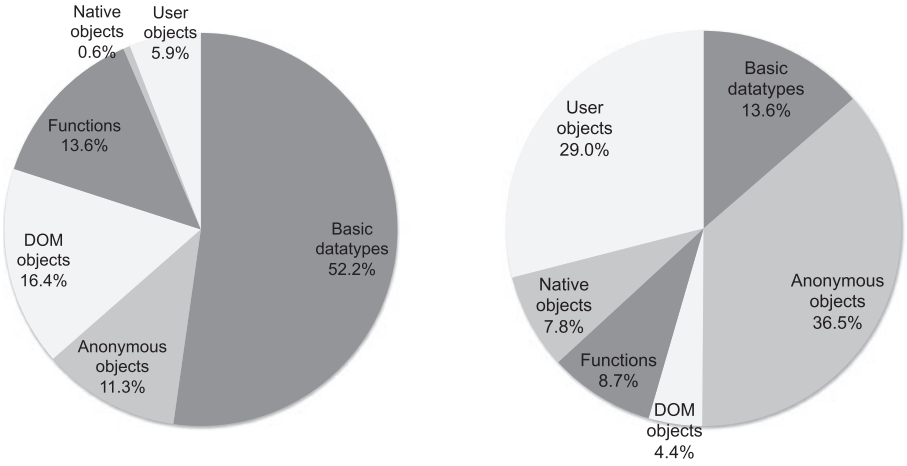
3.1. Design decisions

Benchmarks and tools. JavaScript is the most popular programming language used for client-side web applications. We chose to conduct the experiments on websites implemented in JavaScript to reflect the behavior of real JavaScript applications. In our study, we used the benchmarks collected by Richards *et al.* [13]. The benchmarks consist of 114 dynamic traces extracted from 70 popular websites based on the *Alexa* (http://www.alexa.com/) list. In the benchmarks, most websites were observed in one dynamic trace, while some top websites were explored by several traces (e.g., sites of *google.com* such as *gmail* and *google maps* were presented in 12 separate traces). These traces were collected by an instrumented Safari browser, *TracingSafari* [13], which is capable of recording the dynamically loaded source code and other operations (e.g., writes, deletes, and calls). Our experiments were implemented as an augmented version of an offline analysis tool, *TraceAnalyzer* [13]. In this paper, we focus on analyzing the operations that affect the behavior of JavaScript objects (e.g., property writes and deletes). More details of the experiments are discussed in Section 3.2.

Object categories. During the execution of JavaScript programs, there are different kinds of objects[§] being allocated. We categorize these objects into the following kinds: (i) basic datatypes (i.e., the built-in objects in JavaScript including *Date*, *Array*, *String*, *Regular Expressions*, etc.); (ii) anonymous objects (i.e., the objects created via a pair of braces {...}); (iii) DOM objects (i.e., the HyperText Markup Language (HTML) document objects); (iv) functions (i.e., the objects created by the *Function* constructor); (v) native objects (i.e., the objects created through execution of unavailable native code); and (vi) user objects (i.e., the objects created via the *new* constructor expression either in the application code or in the JavaScript libraries). We defined these object categories based on (i) the dynamic traces and reports from Richards *et al.* [13] and (ii) our observations of and experience with JavaScript programs and objects.

[§]In our discussion, we use the term *object* to refer to an observed object instance during execution and use the term *property* to refer to an observed object property instance during execution.

DOI: 10.1002/spe



(a) Percentage of object instances in each object category.

(b) Percentage of operations associated with each object category.

Figure 1. Object categories and their related operations.

Table I. The relationship between instructions and operation kinds.

Instruction	Operation kind	Preconditions
property write	add	The property does not exist locally or on the prototype chain.
	override	The property does not exist locally but exists on the prototype chain.
	update	The property exists locally.
property read	read-inherit	The property does not exist locally.
	read-local	The property exists locally.
property delete	delete	
constructor return	constructed	This operation only occurs on user objects.

Figure 1(a) shows the distribution of the object instances in these categories over all traces. More than 50% of the instances are basic dataypes among which arrays are the most frequently created. The number of user object instances are relatively small (5.9%), and less than 1% of the instances are native objects. Figure 1(b) shows the distribution of the number of operations[¶] for each object category over all traces. Most operations occurred on anonymous objects (36.5%) and user objects (29.0%), while only 4.4% of the operations occurred on DOM objects. Comparing the results in Figure 1(a) and (b), we observe that native objects, user-defined objects, and anonymous objects were more active than the other object categories (i.e., on average, more operations were associated with each object in these categories.). On average over all the objects of each category in the traces, a native object was associated with 73 operations and a user object was associated with 26 operations; however, either a basic datatype or a DOM object was associated on average with only one operation. Because our goal was to investigate the dynamic behavior of JavaScript objects, we chose the more active object categories (i.e., user objects and native objects) as the focus of this study.[‖**]

Methodology. Previous work [13, 27] defined and discussed the requirements and methodology for designing a dynamic study. This paper adapts some metrics (e.g., object size) from Dufour *et al.*

[¶]In our study, we only consider the operation kinds in Table I. We do not analyze other operation kinds (e.g., function call) recorded by TracingSafari. In our discussion, we use the term *operation* to refer to one of the operation kinds in Table I.

[‖]Ocariza Jr. *et al.* [18] reported that many JavaScript bugs were related to DOM objects. Because DOM objects were associated with a relatively small number of operations, it may be possible to investigate the cause of DOM object related JavaScript bugs easily in future work.

[**]Despite the fact that anonymous objects are relatively active as shown in Figure 1, the results in Figure 3 suggest that anonymous objects are not actively using inheritance and that they experience fewer object property changes. Therefore, we chose not to perform an in-depth study on anonymous objects.

DOI: 10.1002/spe

[27] and is complementary to the study of Richards *et al.* [13], focusing on dynamic object behavior. In Section 4, we present our metrics, illustrating the characteristics of the benchmark applications. In Section 5, we discuss the JavaScript object behavioral patterns found via case studies. The metrics present the summarized behavior, and the specialized cases more intuitively demonstrate representative conditions.

3.2. Experimental setup

A trace is a compressed file containing a source code and a sequence of instructions that are recorded at runtime [13]. In our study, we focused on the following instructions that are related to JavaScript object behavior: (i) property writes; (ii) property reads; (iii) property deletes; and (iv) constructor returns. In the experiments, we analyzed the instructions in sequence and assigned a unique operation kind for each instruction. Table I shows the relation of instructions to operation kinds. Note that the same instruction may result in different operation kinds under different circumstances. In JavaScript, a property write instruction can only change a local property value (i.e., write to the property of the object itself, not to property of its prototype object). In our analysis, a property write may result in one of the three operation kinds (i.e., *add*, *override*, and *update*) for a better understanding of its effect on object properties and inheritance. The property lookup mechanism in JavaScript, on the other hand, may use the prototype chain to read an inherited property; hence, we assign one of the two operation kinds (i.e., *read-inherit* and *read-local*) to a property read instruction. We use the *constructed* operation for a user-defined object to distinguish its construction stage from the rest of its object lifetime.

We modified *TraceAnalyzer* to produce the operation kinds in Table I and implemented an object category filter to focus on the object categories that best demonstrated dynamic behavior (Section 3.1). Our implementation produced both aggregated results and detailed information for individual objects. Figure 2 shows an example of the history information (i.e., sequence of operations) associated with an object. For each operation, we output the operation kind, property name, property type, and other information (e.g., the property access chain for a *read-inherit* operation). We used such information to study object behavioral patterns (Section 5). The experimental results were obtained in a 2.66 GHz Intel Core 2 Duo MacBook Pro with 4 GB memory running the Mac OS X 10.6.8 operating system.

3.3. Threats to validity

There are several aspects of our empirical study which might threaten the validity of our conclusions: (i) Although the websites we used were listed at *Alexa* as most popular 5 years ago, we cannot know how representative the input is of current website usage. In addition, websites today may exhibit

```
======================Object Information======================
609. Object ID: 15796
     Category: user object
     Operations:      24
     Prototype: 8054

-----------History-----------
1. OpKind: add        Property: fn                Type: 9697(function)  OpID: 67849
2. OpKind: add        Property: obj               Type: 15261(unknown)  OpID: 67852
3. OpKind: add        Property: overrideContext
   Type: 15104(constructed by 9807(function))     OpID: 67863
4. OpKind: constructed
......
21. OpKind: read-inherit     Property: contains   Type: 8057(function)
    Chain: 15796-8054        OpID: 329322
22. OpKind: read-local       Property: fn         Type: 9697(function)  OpID: 329323
23. OpKind: delete           Property: fn         OpID: 329327
24. OpKind: delete           Property: obj        OpID: 329328
```

Figure 2. Sample history information of an individual object from *yahoo.com*.

DOI: 10.1002/spe

different behaviors from what we observed. We plan to conduct a study on web application evolution in future work. (ii) The traces may not cover all possible behaviors of the websites; thus, there may be important behavior not explored by the dynamic trace collection. (iii) The traces were collected by executing websites under one browser (i.e., Safari). JavaScript code specific to other browsers might not be executed so that it was not collected and analyzed in our study.

4. METRICS

In this section, we present the metrics for summarizing the characteristics of all object categories, user objects, prototype-based inheritance, and native objects.

4.1. General metrics of JavaScript objects

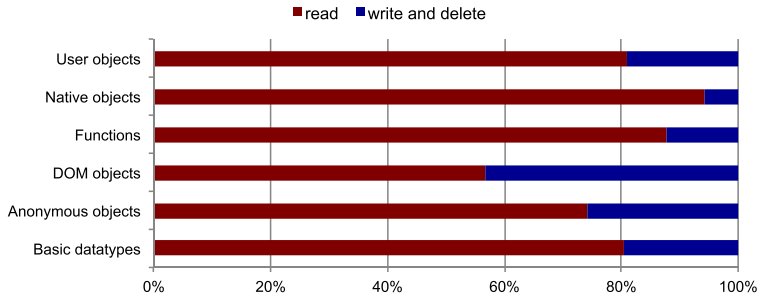
We first discuss the metrics gathered for all object categories.

Operation kind distribution. The behavior of an object is defined by its associated operations. An object is more dynamic when its properties change (e.g., *override*, *add*, or *delete*) frequently. The percentage of *read-inherit* operations suggests the importance of precisely knowing the prototyping mechanism. Figure 3 shows the operation kind distribution of objects in all categories over all traces. Figure 3(a) presents the distribution of read (i.e., *read-local* and *read-inherit*) versus write and delete (i.e., *add*, *update*, *override*, and *delete*) operations. For each object category, except for DOM objects, read operations comprised at least 75% of all operations, indicating a relatively small fraction of operations may possibly change properties. In contrast, DOM objects were associated with many write or delete operations (about 45%), because interactive JavaScript webpages tend to make frequent changes to the DOM for updated content such as forms.

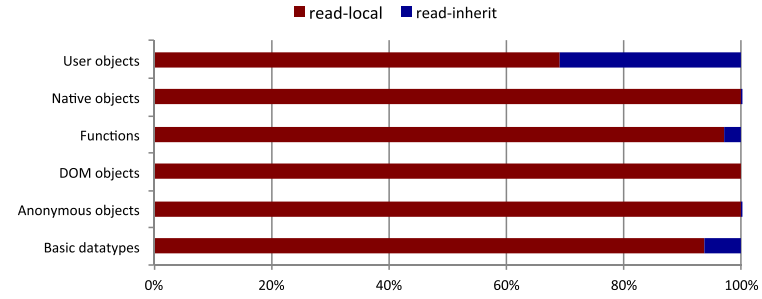
Figure 3(b) and (c) presents additional details on the information in Figure 3(a). Figure 3(b) shows that *read-local* operations dominated the read operations for most object categories. The DOM objects never experienced a *read-inherit* operation (i.e., use of a prototype chain to read an object) because HTML nodes (e.g., elements and attributes) can be directly accessed (i.e., *read-local*). Similarly, the anonymous objects also had very few *read-inherit* operations (278 out of 3.3×10^7 operations). On the other hand, more than 30% read operations of user objects were *read-inherit* operations, suggesting user objects actively use their prototype chains to lookup properties.

Figure 3(c) illustrates the distribution among write and delete operations. *delete* operations occurred in all object categories despite their relative infrequency. *delete* operations comprised 6% of the write and delete operations on basic datatypes, with many instances on array datatypes. User objects also experienced relatively many *delete* operations (i.e., 3% of all write and delete operations), which means properties of user objects are sometimes removed at some point during execution. Another infrequently observed write operation was *override*. Recall that we count a property write instruction as *override* only when the property does not exist locally but exists on the prototype chain. Less than 0.1% of the write and delete operations of basic dataypes, anonymous objects, DOM objects, and native objects were *override* operations. Even so, *override* operations occurred more often on functions and user objects; specifically, 10% and 14% of write and delete operations of functions and user objects were *override* operations, respectively. *add* and *update* operations were much more frequently observed, with different distributions for each object category. Native object was the only object category that contained more *update* operations than *add* operations. The code that initialized native objects was not observed by *TracingSafari* because it was not written in JavaScript. Intuitively, the initialization stage of an object consists mostly of *add* operations; this suggests why there were fewer *add* operations recorded for native objects. For other object categories, at least 60% of write and delete operations were *add* operations, and at least 20% of write and delete operations were *update* operations. To sum up, based on the operation kind distribution in Figure 3, user objects exhibit the most dynamic characteristics (i.e, read instructions incur many prototype chain lookups and all the four property changing operation kinds are frequently observed). We will provide more detailed discussion on property changes of user objects in Section 4.2.

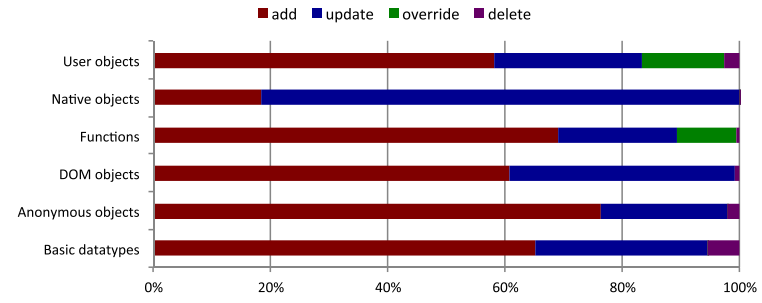
DOI: 10.1002/spe



(a) Read vs. write and delete operation distribution.



(b) read-local and read-inherit operation distribution.



(c) Write and delete operation distribution.

Figure 3. Operation kind distribution for all object categories. DOM, Document Object Model.

Discussion. The *add*, *override*, *update*, and *delete* operations of a JavaScript object may change its behavior at runtime. For a static program analysis to better model JavaScript objects, it is important to propagate property changes precisely via flow sensitivity (i.e., following the execution order of statements in a program). However, a fully flow-sensitive analysis may be too costly to be practical for analyzing large JavaScript programs [22]. Figure 3(a) shows that the majority of JavaScript object operations are property reads, which do not change object behavior. Thus, it is possible to do a partially flow-sensitive analysis to enable strong updates at statements that may change object properties, a small fraction of all statements in JavaScript web applications. The data suggest that this will result in a scalable analysis. Such a partially flow-sensitive analysis was presented recently [21].

Number of operations. Figure 4 shows a distribution of the number of operations experienced by each object having at least one operation in the traces. Most of the objects had a relatively small number of operations as 25% of the objects were associated with one to two operations, and 75% of the objects had no more than eight operations. However, there still were many objects associated with a relatively large number of operations (i.e., 1.5% of the total 70,723 objects were associated with at least 100 operations). The two extreme cases had more than 1×10^6 operations. Despite differences between websites in the extreme numbers of operations associated with an object, all the websites contained relatively few objects with large numbers of operations. We will compare Figure 4 with the same metric applied only to user objects in Figure 6 in Section 4.2.

DOI: 10.1002/spe

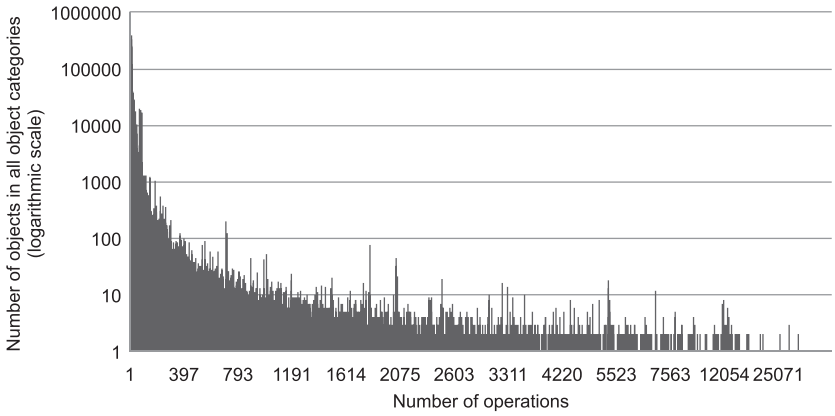


Figure 4. Number of operations distribution for all objects.

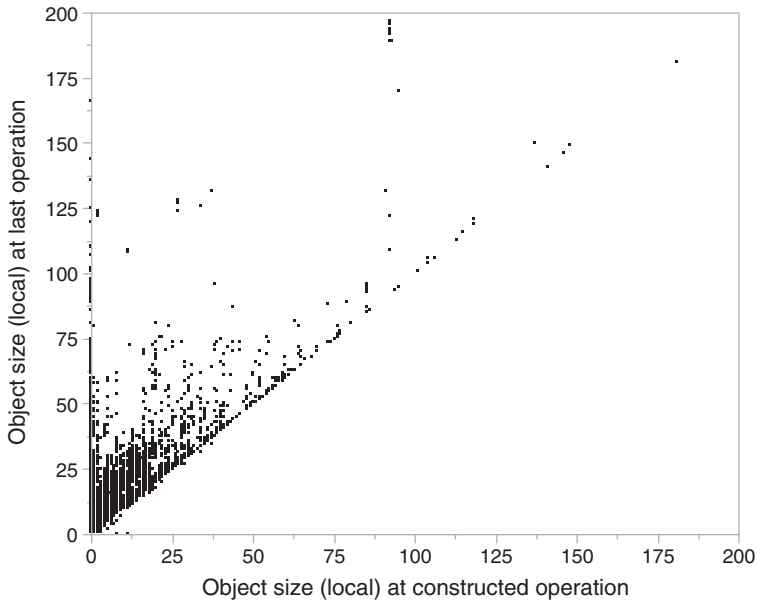


Figure 5. Local sizes of user objects at their *constructed* and last operations.

4.2. User objects

We now focus on studying the characteristics of the user objects.

Object size. Recall that the behavior of a JavaScript object is decided by its associated properties. We define the size of a JavaScript object as its number of accessible properties (including local and inherited properties) at a program point during execution. Because the property list of a JavaScript object is not fixed at runtime, object size may change. We calculate JavaScript user object sizes at two crucial stages in object lifetime: at its *constructed* operation and its last observed operation (i.e., an approximation of the end of object lifetime). On average over all the user objects, the object size was 28 at the *constructed* operation. Figure 5 shows the local sizes of user objects (i.e., counting only local properties) at their *constructed* and last operations. There were many user objects whose local sizes were the same at both operations in their lifetime (i.e., the points on the $x = y$ line). However, we observed that the local sizes of many user objects grew significantly by the end of their lifetime compared with local sizes at their *constructed* operations. This result gives evidence that the local size of a JavaScript user object is usually not consistent at different stages of its lifetime and in most cases increases. We plan to investigate object property changes throughout its lifetime in future work.

DOI: 10.1002/spe

Discussion. Static program analysis often identifies objects by their creation site (i.e., all objects created by the same statement are represented by the same abstract object) [31]. The precision of analysis results is dependent on the object representation choice. Based on the fact that a JavaScript constructor may be polymorphic [13] and the object size changes over its lifetime, the creation site alone may not be representative of a set of JavaScript objects. More accurate object representations (e.g., a representation that identifies objects by their creation site as well as the local properties [21]) are needed for better analysis of JavaScript objects.

Number of operations. Figure 6 shows the distribution of the number of operations experienced by each user object with at least one operation in the traces. We observed that user objects usually had more operations than objects in other categories. Specifically, 25% of the user objects were associated with fewer than six operations, while 25% of the objects in the other categories experienced only one operation. Furthermore, 75% of the user objects had no more than 13 operations, while 75% of the objects in the other categories experienced at most six operations. We also found that user objects were most likely to be associated with at least 100 operations (i.e., 1.9% of user objects were associated with at least 100 operations, and only 0.2% of the objects in the other categories were associated with at least 100 operations). This result supports our previous observation that user objects behave more actively than other object categories.

Property type changes. Recall that JavaScript allows an object property to be changed at any program point (Section 2.1); moreover, the type of an object property may be changed via property write instructions. If the type of a property is altered, the behavior of the object will very likely

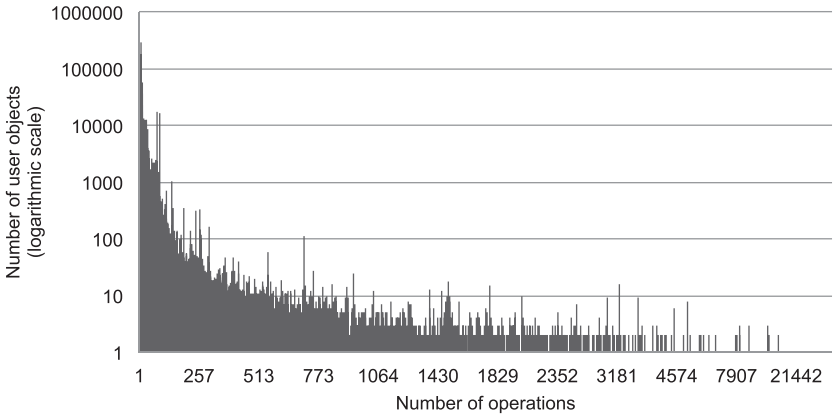


Figure 6. Number of operations distribution for user objects.

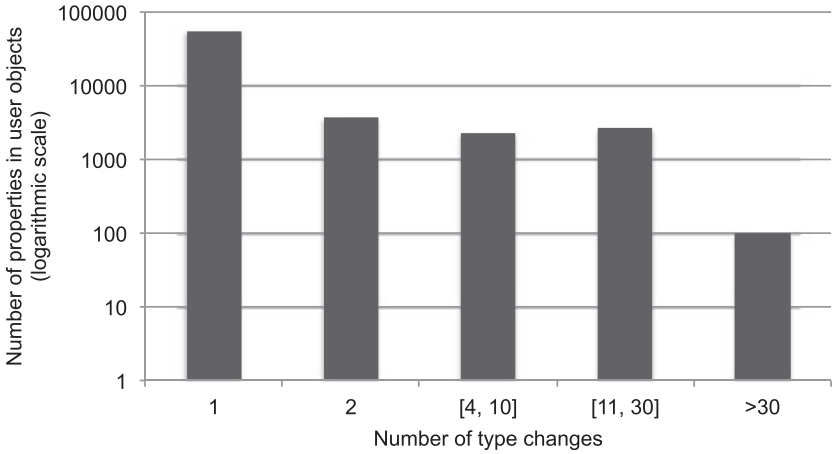


Figure 7. Number of type changes distribution for user object properties.

DOI: 10.1002/spe

be changed as well. In Figure 7, we show the distribution of the number of type changes for all properties with at least two types. We define the following as different property types: (i) every JavaScript basic datatype; (ii) every constructor of user objects; and (iii) every instance of the rest of the object categories in Figure 1. For example, if an *update* operation results in a change in value of reference property *o.p* from user object o_1 to user object o_2, where o_1 and o_2 are different object instances created by different constructors, we calculate that *o.p* may have two different types at different program points. We used the constructor of a user object to approximate its type because this was less costly to check than the actual type equivalence. In JavaScript, property changes from one object instance to another can result in a very different behavior. Interesting cases and patterns of property type changes are presented in Section 5. Over all the traces, 99% of the properties in the user objects did not change types, suggesting most properties were implemented with fixed types. Nevertheless, there were many properties (i.e., 64,721 of the properties in user objects) whose type changes, and some of them changed surprisingly many times. As shown in Figure 7, more than 50,000 properties have two different types (i.e., change type once) during the execution. Almost 3000 properties underwent more than 10 changes in their lifetime, and one of them changed more than 10,000 times (i.e., a property named *_next* from *me.com*). The results suggest that the developer may be using these properties as temporaries for implementing different functionalities at different program points.

It is possible that a large amount of property type changes do not result in as many types of the property. For example, a property that changes consistently between *String* and *Boolean* has many property type changes but only two types. Nevertheless, manual inspection of our data show that in most cases, a property type change introduces a new type for the property.

Discussion. Most properties of user objects never change. This result is consistent with the assumption made by Schäfer *et al.* [25], suggesting that these techniques (i.e., determinacy analyses [25, 26]) are promising to use to optimize most properties in JavaScript programs. Nevertheless, it was reported by Sridharan *et al.* [32] that some property accesses (i.e., both read and write) render static analysis for JavaScript inaccurate and unscalable on real applications. These authors observed that many property accesses are correlated (i.e., a dynamic property *r* and a property write *w* are correlated if *w* writes the value read at *r*, and both *w* and *r* must refer to the same property name). Specialized static analysis techniques (e.g., program transformation and context-sensitive analysis [32]) can be applied to the property accesses that exhibit correlations, especially for the properties that are frequently accessed shown in Figure 7, to obtain better precision and scalability.

Object instances versus constructors. Our study is a dynamic analysis that reflects the behavior of JavaScript objects in the observed executions. User objects are created by calling constructor functions. We present the relationship between the created user objects and the constructors from *facebook.com*, *google.com*, and *yahoo.com* traces in Figure 8. For each of the three websites, we show the number of singleton constructors, the number of non-singleton constructors (i.e., constructors that created more than one observed instance), and the number of user object instances, respectively. Most constructors (on average 86% over the three websites) created only one object instance, ranging from 63% (*facebook.com*) to 90% (*google.com*). However, on average over these websites, each constructor instantiated 12 objects, and some of the constructors generated a large number of instances (e.g., one constructor function in *facebook.com* created 5901 objects).

Discussion. The existence of many singleton constructors presents opportunities for just-in-time optimization [11] for JavaScript objects. Trace-based information may be used to specialize the representation of the objects created by singleton constructors. In addition, the observation of other constructors generating many object instances suggests that it may be important to accurately model these constructors via advanced static analysis techniques (e.g., recency abstraction [33]).

4.3. Prototype-based inheritance

In this section, we summarize the observed characteristics of prototype-based inheritance for user objects. All JavaScript objects (except for *Object.prototype*) may inherit properties from their prototype objects. JavaScript object inheritance is decided at runtime by the constructor and also can be changed at any program point.

DOI: 10.1002/spe

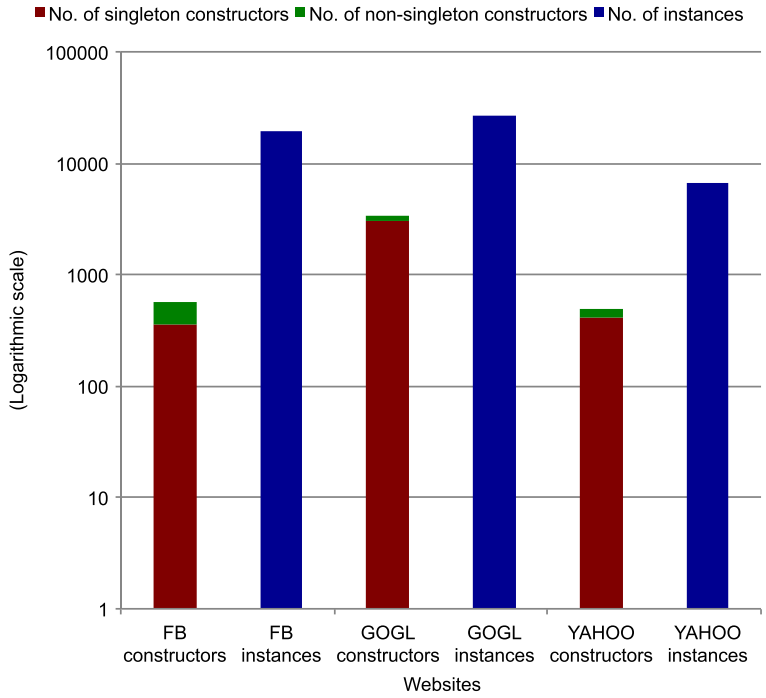


Figure 8. User object instances and constructors.

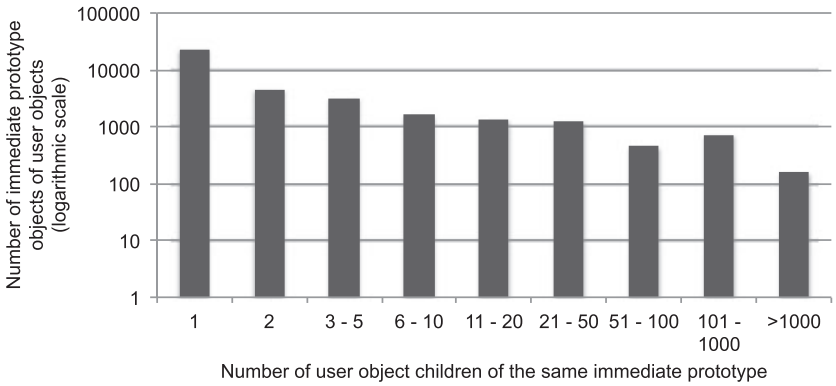
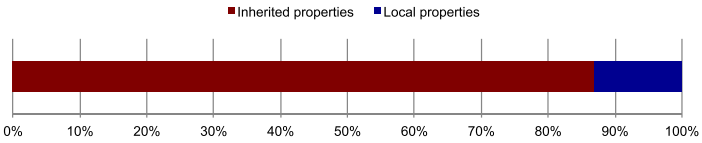


Figure 9. Characteristics of prototype object reuse.

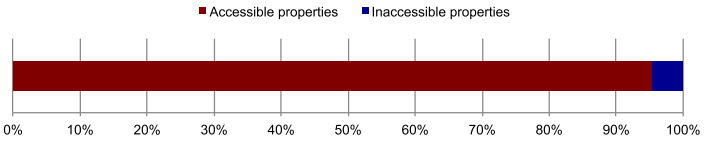
Prototype object reuse. At the end of each user object's lifetime, we observed its immediate prototype object. We then grouped all user objects with the same immediate prototype object and recorded the number of objects in the group, k. Figure 9 shows the number of prototype objects with k immediate user object children. We observed that many of the objects (64%) were used as the immediate prototype for only one user object. These results were not expected because inheritance is usually thought of as a mechanism for code reuse. Because we only consider immediate prototype objects in Figure 9, it is possible that many user objects inherit properties from a prototype object that is higher in their prototype chains. Over all the immediate prototypes of user objects, a prototype object had on average of 35 immediate children.[††]

[††] *Object.prototype*, the default prototype object, is always at the root of any prototype chain in JavaScript and is included in the average. Many of the prototype objects in the rightmost bar in Figure 9 are instances of *Object.prototype* from particular traces.

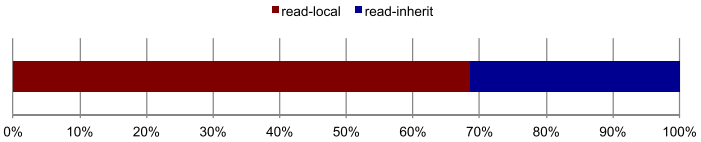
DOI: 10.1002/spe



(a) Inherited vs. local properties in the user objects.



(b) Accessible vs. inaccessible properties from prototype objects.



(c) Number of read-local operations vs. number of read-inherit operations.

Figure 10. Property inheritance and reads. The results of (a) and (b) were measured at *constructed* operations; the results of (c) were measured during executions.

Discussion. Prototype inheritance is not easy to analyze via static analysis because the delegation model allows the inheritance of an object to be changed at runtime. It is expensive to maintain accurate prototype chains during static analysis. Our results in Figure 9 suggest that the actual inheritance of JavaScript objects is not complicated in most cases. For a prototype object that is used as the immediate prototype for only one user object, it is possible to simplify the inheritance structure by 'inlining' the inherited properties (i.e., because a prototype object is a 'singleton' prototype; the properties visible from the child can be treated like local properties during static analysis, such that the analysis avoids maintaining prototype chains and performing property lookups). This idea may be applied to more than half of the prototype objects (according to Figure 9) using feedback-directed optimization techniques. This may help increase both precision and performance.

Property inheritance and reads. Inherited properties serve as the goal for code reuse, while overridden properties allow more specific behavior of objects. Figure 10(a) shows the percentage of inherited and local properties of all the accessible properties in user objects at their construction stage (*constructed* operations). For each user object at its *constructed* operation, we collected the local property list and the property lists of its prototype objects. All properties in the local property list were counted as local properties, and a subset of the properties in the property lists of its prototype objects were counted as inherited properties conforming to the JavaScript property lookup mechanism. We found that 13% of the properties were implemented for specific user objects, while most of the properties (87%) were inherited from prototype objects.

Over all the properties in the prototypes of user objects, Figure 10(b) shows the percentage of properties accessible from user objects (at the *constructed* operations). For the prototype objects of each user object at its *constructed* operation, properties that were overridden were counted as inaccessible properties; others were accessible properties. In Figure 10(b), 5% of the properties were overridden so that they were inaccessible, and the rest (95%) were accessible. We observed that objects higher in the prototype chains (closer to *Object.prototype*) often were associated with more properties. Our hypothesis from this observation is that developers of JavaScript websites intend to build/use large prototype objects allowing other objects to inherit properties from them.

The behavior of an object is reflected by the read operations associated with it. Figure 10(c) shows the percentage of observed *read-local* operations and *read-inherit* operations of user objects. About 69% of the recorded read instructions did not require prototype object lookup (i.e., *read-local*). Comparing with the results in Figure 10(a), we conclude that although the majority of the properties of a user object may be inherited from its prototypes (at *constructed* operation), during execution, local properties are read more often than inherited properties.

DOI: 10.1002/spe

Discussion. This result suggests that local properties may be more important to model accurately and that there will be fewer local properties than inherited ones. The context-sensitive analysis presented in Wei and Ryder [21] used a calling context that accurately reflected the properties of the receiver object. Local properties and the prototype objects (but not the inherited properties) were used in the calling context approximation; our results in Figure 10 support this analysis choice.

4.4. Native objects

We now study the characteristics of the native objects.

Native object statistics. It is important to understand the behavior of native objects in JavaScript code because for most software tools built for JavaScript, only partial information for a native object is available (i.e., the code that creates a native object is not available). Figure 11 shows the percentages of native objects not associated with any operation (i.e., the first six operations in Table I omitting *constructed*), associated with only read operations (i.e., *read-local* or *read-inherit*), and associated with write or delete operations (i.e., *add*, *override*, *update* or *delete*). Unlike user objects, most native objects (90.5%) were inactive (i.e., without any operations on their properties). Note that these objects may be associated with other operations not analyzed in this paper (e.g., calls). For the rest of the native objects with operations, almost half of them were associated only with read operations, suggesting that their properties remain consistent. Only 4.4% of native objects experienced property value changes.

Discussion. This result suggests that it is possible to model a native object via its uses because of its almost constant property set. Madsen *et al.* presented a use analysis to inference the behavior of native objects [6]. The assumptions made by the authors (e.g., properties are not dynamically added or removed after the object has been fully initialized) are reasonable for native objects based on our empirical findings.

Number of operations. Figure 12 shows the distribution of the number of operations experienced by native objects having at least one operation in the traces. Specifically, 25% of these native objects were associated with one to three operations (similar to the result in Figure 4 for all object categories), and 75% of these native objects had no more than 49 operations (significantly more

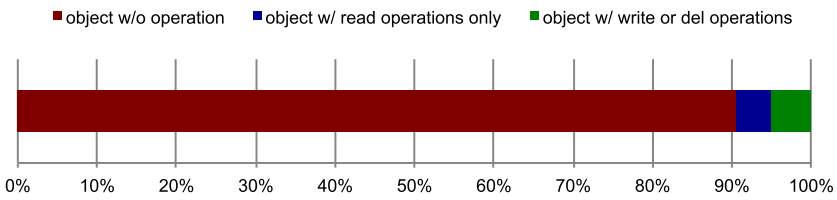


Figure 11. Native object operations.

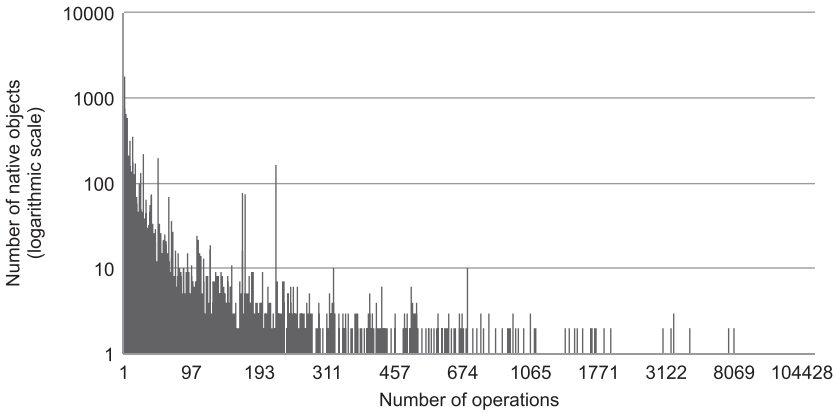


Figure 12. Number of operations distribution for native objects.

DOI: 10.1002/spe

than the result in Figure 6 for user objects). This result shows that although most native objects were associated with few operations, there are some specific native objects that are active.

5. JAVASCRIPT OBJECT BEHAVIORAL PATTERNS

In this section, we present a study on the object behavioral patterns across the benchmarks, and then discuss some representative cases. An operation occurrence pattern illustrates a representative sequence of operations occurring on a specific object or property. We studied those operation occurrence patterns that pose challenges for building software tools or that help to better understand coding and usage practices of JavaScript objects. We also observed unexpected operation occurrence patterns. A property type change pattern presents frequently occurring type changes from one object category to another on specific properties. We investigated cases of property type change patterns that are not commonly understood.

5.1. Operation occurrence patterns – user objects

Recall that our analysis outputs history information (i.e., the sequence of operations) for an object. The re-occurring sequence of operations on objects or properties may suggest a programmer's coding style, familiarity with JavaScript, and/or good/bad programming practice. The occurring sequence of operations on an object may result not only from programming style but also from user interactions (e.g., for user objects, the operations after construction may be triggered by user events). Knowing the operation occurrence patterns, we can better investigate the challenges of understanding JavaScript programs. In this section, we discuss studies that find behavioral patterns in the traces from more than one million user objects and more than nine million of their properties.

Table II shows the interesting operation occurrence patterns we studied. Each pattern consists of at most three different operation kinds. We designed these patterns based on the following: (i) operation sequences frequently observed in the experiments and (ii) the usage of delegation in JavaScript. Patterns 1–5 reflect the sequences of operations on a specific property *p*, and patterns 6–9 show the relationship between a *constructed* operation and other operations on a user object. We use two quantifiers to express the number of times an operation occurs (i.e., + for operations occurring 1 or more times and *{n}* for operations occurring n times exactly). We discuss each pattern and our empirical observations in detail in the succeeding discussions.

Operation occurrence patterns on a specific property. Because *delete* is not supported by most popular programming languages, its semantics have not been widely studied. In addition to property deletions that remove local properties, property write operations (i.e., *add*, *override*, and *update* operations) may also add/change the values of local properties. We present five operation occurrence patterns to summarize unexpected or interesting sequences of operations experienced by JavaScript properties.

First, we study the relationship between write operations and *delete* operations.

- Pattern 1: *(add p | override p | update p)+ → delete p*. This pattern occurred on 106,137 properties of the user objects in 56 out of 114 traces in the benchmarks. The average percentage of properties of user objects exhibiting pattern 1 in the 56 traces was 0.9% with a standard

Table II. Operation occurrence patterns of user objects.

	Operation occurrence pattern	Notes
1	*(add p | override p | update p)+ → delete p*	regular and abnormal *delete* practices
2	*(add p | override p | update p){0} → delete p*	
3	*read-local p+ → delete p → read-inherit p+*	local and inherited property lookups for same *p*
4	*read-inherit p+ → override p → read-local p+*	
5	*read-local p+ → update p*	'temporary' property
6	*delete+ → constructed*	*delete(s)* before/after *constructed* operations
7	*constructed → delete+*	
8	*update+ → constructed*	interesting writes before/after *constructed*
9	*constructed → (add | override)+*	

DOI: 10.1002/spe

deviation of 1.8% across the traces. With this relatively high standard deviation, we observed the traces that contributed most to this pattern were from *google.com*, especially *gmail* (i.e., 9% of properties of user objects in *gmail*, 62,055 in total). The scenarios of pattern 1 can be interpreted as follows: (i) if the pattern occurs only a few times, the specific property is accessible only between the property write operation and the *delete*; (ii) if the pattern occurs on the same property many times, the property is most likely used as a temporary variable. The majority (99.8%) of properties that exhibit pattern 1 only contain one iteration of the pattern. This suggests that most uses of *delete* are to end the lifetime of a specific property; after the local property is deleted, an inherited property (if exists) will be accessible. Nevertheless, there are some properties exhibiting multiple occurrences of pattern 1. For example, *pattern* 1 occurred 149 times on one property of a user object in *mozilla.com*; this property *lastAction* is used to check if a function may be called and then it is deleted. In this scenario, the developer creates the property when needed and uses a delete statement to ensure that specific property only exists in a certain part of the program. This usage is considered as a legitimate use of *delete*.

- Pattern 2: *(add p | override p | update p){0}* $\rightarrow$ *delete p*. This pattern occurred on 70,143 properties of the user objects in 37 out of 114 traces in the benchmarks. The average percentage of properties of user objects exhibiting pattern 2 in the 37 traces was 0.9% with a standard deviation of 1.6%. The distribution of pattern 2 is similar to pattern 1 in that *google.com* dominates the uses of the *delete* operation with 7% of the properties of user objects (i.e., 45,440 in total) in *gmail* exhibiting pattern 2. This pattern occurs more frequently than expected because *pattern* 2 describes that a property *p* is deleted when it never had been added, overridden, or updated. In JavaScript semantics, a *delete* operation on a non-existing local property does not alter the object. The developers of JavaScript websites are likely using the *delete* statement to ensure that a local property does not exist at some program point. Although the occurrence of *pattern* 2 during execution will not produce a runtime error, it reflects the difficulty of controlling properties of a JavaScript object. We regard this pattern as a bad practice because it increases the number of non-meaningful operations in JavaScript programs and may adversely affect program understanding. We would like to further investigate the frequent uses of *delete* with *google* in future work.

After a local property *p* is deleted from the object *o*, reading *o.p* uses the prototype chain of *o*. The following pattern shows that at different points of the execution, a *delete* operation may result in accessing local versus inherited properties.

- Pattern 3: *read-local p+* $\rightarrow$ *delete p* $\rightarrow$ *read-inherit p+*. This pattern occurred on 311 (0.03‰) properties of the user objects in the benchmarks. Although this pattern does not occur as frequently as the others, it is the most straightforward pattern showing the influence of *delete* operation on the uses of an object property. The occurrences of pattern 3 are limited to fewer than 10 websites (e.g., *npr.org*), and the deleted properties are all function properties. Perhaps, this implies specialization of the function properties; a read operation on the object results in the use of a different property lookup mechanism at different program points (i.e., local property versus prototype chain lookup). The Use of pattern 3 definitely poses challenges to understanding the behavior of JavaScript programs.

The *override* operations also affect the local property list of an object such that reading a property of an object may result in *read-inherit* versus *read-local* operations before and after the *override* operations, respectively. In addition, because an *update* operation changes the value of a local property, the *read-local* operations at different program points may return different results if an *update* operation occurs between them.

- Pattern 4: *read-inherit p+* $\rightarrow$ *override p* $\rightarrow$ *read-local p+*. This pattern occurred on 281,160 properties of the user objects in 73 out of 114 traces in the benchmarks. The average percentage of properties of user objects exhibiting pattern 4 in the 73 traces was 1.6% with a standard deviation of 3.2%. The two websites that experienced significant number of pattern 4 were *me.com* (i.e., an Apple online services site when the data were collected; it is now replaced by

DOI: 10.1002/spe

iCloud) and *npr.org*. This pattern directly shows the impact of an *override* operation on the uses of the property, and it occurs more frequently than pattern 3. Similar to the pattern 3, pattern 4 indicates that understanding JavaScript property accesses can be difficult, requiring software tool support.

- Pattern 5: *read-local p+* $\rightarrow$ *update p*. This pattern occurred on 616,825 properties of the user objects in 106 out of 114 traces in the benchmarks. The average percentage of properties of user objects exhibiting pattern 5 in the 106 traces was 7% with a standard deviation of 5.5%. As the most frequently observed pattern on a specific property in the benchmarks, pattern 5 exists in almost all the websites. *go.com*, *facebook.com*, and *yahoo.com* are the three websites with the highest percentage of properties of user objects experiencing this pattern (i.e., 41%, 23%, and 21%, respectively). We observed that there were 127,052 properties in the benchmarks that were read and updated more than once. If *update* happens frequently on a property, then this property may be regarded as a temporary variable. For example, the property *_next* of an object in *me.com* exhibits 12,620 iterations of this pattern; this property is used to build a data structure similar to a linked list. Frequently updating a property value is a common object-oriented practice for building data structures (e.g., list) and control structures (e.g., loop) in the program, while updating the type of a property is unusual. In Section 5.2, we further study the property type change patterns observed in the traces.

Operation occurrence patterns related to the *constructed* operations. We divide the lifetime of a user object into two stages: before and after construction. Different object behavior is expected at these two stages: (i) before a user object is constructed, properties should be frequently added/overriden and (ii) after a user object is constructed, its properties often may be used and updated. Richards *et al.* [13] discussed object protocol dynamism (i.e., operation distribution throughout object lifetime) on some representative websites, in which an object lifetime was also divided by the construction phase. The results reported in Richards *et al.* [13] focused on the overall aggregate object behavior of websites. In our study, we present results on individual objects and summarized more generalized conclusions across all benchmarks.

We first investigated the stage of a user object (before or after *constructed*) at which *delete* operations happen.

- Pattern 6: *delete+* $\rightarrow$ *constructed*. This pattern illustrates that property deletion occurs in the construction stage of an object (i.e., the deleted property may not be accessed after the object is constructed). It occurred on 34,725 user objects in 23 out of 114 traces in the benchmarks. The average percentage of user objects exhibiting pattern 6 in the 23 traces was 9% with a standard deviation of 12%. Traces from *google.com* (including *gmail* and *google docs*) all contained objects that experienced many occurrences of this pattern, from 18% to 51% of user objects in each trace. Other websites that frequently exhibit pattern 6 (more than 5% of user objects) are *virtualsecrets.com* and *nor.org*. There is a strong correlation between pattern 6 and pattern 2. Most of the deleted properties within the construction stage do not exist locally at the time of deletion. For example, an object from *myspace.com* exhibits seven deletions of the same property *q* in its constructor, although the property never exists locally. After inspecting the code, we found that when several function properties of the object are defined, the delete statement (i.e., *delete this.q*) will always execute without checking the existence of the property *q*. Along with pattern 2, we hypothesize the following two scenarios explaining this practice: (i) developers use the *delete* to ensure a specific property does not exist at a certain program point, not knowing for sure if the property exists and (ii) as JavaScript software evolves, a delete operation that used to be meaningful becomes useless, and it is not noticed or removed because it does not affect program behavior.
- Pattern 7: *constructed* $\rightarrow$ *delete+*. The occurrence of the *delete* operation after the construction stage is considered normal if the property is used before the deletion (e.g., pattern 1). Pattern 7 occurred on 42,372 user objects in 56 out of 114 traces in the benchmarks. The average percentage of user objects exhibiting pattern 7 in the 23 traces was 4.3% with a standard deviation of 9%. Pattern 7 was observed in a larger set of websites than pattern 6. In addition to *google.com*, many user objects from *facebook.com* also exhibited pattern 7. We observed that

DOI: 10.1002/spe

deletions happen frequently in some objects; more than 500 objects are associated with at least 50 *delete* operations after the construction stage. Potentially, these objects exhibit very different behaviors in between these *delete* operations. We also observed that among all the user objects exhibiting pattern 7, 17,655 of them had a *delete* operation before usage of a specific property. Discussions of pattern 2 addressed these occurrences.

Property addition and overriding are common for an object within the construction stage. We conducted further study on the update operations in the construction stage and on the add/override operations that happen after an object is constructed.

- Pattern 8: *update*+ → *constructed*. This pattern occurred on 60,742 user objects in 90 out of 114 traces in the benchmarks, much more widely observed than the *delete* operation related patterns (i.e., patterns 6 and 7). The average percentage of user objects exhibiting pattern 8 in the 90 traces was 5.5% with a standard deviation of 6.7%. Websites that most frequently experienced pattern 8 were *yahoo.com*, *go.com*, and *me.com*. Updating properties in the construction stage suggests that some JavaScript object constructor functions do not just create properties. For example, an object from *maps.google.com* contains 764 *update* operations at the construction stage and all its operations (more than 2800) occurred at the construction stage. Updating a property within a constructor is not considered to be a good practice unless it is necessary for initializing data structures, because then the original value of the property is not accessible when object construction finishes. We hope to study pattern 8 more to distinguish good and bad programming practices in future work.
- Pattern 9: *constructed* → *(add | override)*+. This pattern occurred on 265,224 (20.56%) user objects in 106 out of 114 traces in the benchmarks. The average percentage of user objects exhibiting pattern 9 in the 106 traces was 28% with a standard deviation of 23%. For some websites (i.e., *easychair.org*, *raphaeljs.com*, and *me.com*), most user objects (more than 90%) experienced pattern 9. Thus, the local property lists of many user objects are expanded by adding a new property or overriding an inherited property. This result conforms to our observations in Figure 5 and indicates that the behavior of a JavaScript object cannot be safely approximated by its properties at the point of its construction. The presence of this pattern demonstrates the fundamental difference between a JavaScript object and a pre-defined class-based object, illustrating the flexibility of JavaScript programs.

5.2. *Property type change patterns – user objects*

We have observed that the property types of user objects change very often (Figure 7). In this section, we investigate frequently occurring property type change patterns for user objects.

Table III shows the 10 most frequent property type change patterns. Recall that in our study, if a property value changes from referring to one user object to referring to another user object that is created by a different constructor, we assume a type change occurs because these user objects may exhibit significantly different behavior. For functions and anonymous objects, we count a property change from one instance to another as a property type change. For the basic datatypes, on the other hand, a property type change alters one basic datatype to another (e.g., *String* → *Boolean*). It was expected that the top three most frequently occurring patterns would be changes between the same object categories, with the *anonymous object* → *anonymous object* occurring most often. However,

Table III. Property type change patterns.

Pattern	Occurrences	Pattern	Occurrences
anonymous → *anonymous*	59674	*anonymous* → *basic*	3138
user → *user*	43223	*basic* → *user*	1320
function → *function*	34104	*basic* → *anonymous*	260
function → *user*	6384	*basic* → *function*	144
basic → *another basic*	3726	*user* → *function*	46

DOI: 10.1002/spe

the *function → user object*, *anonymous → basic datatype*, and *basic datatype → user object* patterns all occurred more than 1000 times in the benchmarks. These property type changes suggest that many properties are used for unrelated purposes at different program points, another coding practice that poses challenges for understanding JavaScript programs. We observed all possible property type change patterns among basic datatypes, user objects, functions, and anonymous objects (except for *anonymous → function*).

As for individual websites, the five websites we observed with the most property type change patterns of user objects were *google.com* (*gmail* and *google maps*), *280slides.com*, *me.com*, *facebook.com*, and *flapjax-lang.org*. Among the websites where we observed significant number of property type change patterns (i.e., more than 100 occurrences), *anonymous → anonymous* and *user → user* patterns occurred most frequently on *me.com* and *google.com*, respectively. Most *function → function* and *function → user* patterns were from *ebay.com*, while most *basic → another basic* and *anonymous → basic* patterns were from *flapjax-lang.org*. For the less frequently observed patterns, *flapjax-lang.org*, *me.com*, and *google.com* dominated the occurrences of *basic → anonymous*, *basic → function*, and *user → function* patterns, respectively. Based on the results previously discussed, we observed that *google.com* and *me.com* were applying quite flexible object property changes in their code.

We studied some interesting cases among the property type change patterns. Two properties (i.e., *initialActiveChats* and *initialFocusedChat*) of a user object from *facebook.com* change their types from *anonymous* to *Boolean* and from *String* to *Boolean*, respectively. After manually inspecting the source code, we observed that these two properties were set to meaningful values (e.g., *this.initialActiveChats = activeChats*) when used as parameters for calling a function (*this._loadInitialTabs.bind(this, this.initialActiveChats, this.initialfocusedChat)*) and then set to *false*. These properties are only used at certain points of the program and instead of being deleted, the developers kept them as Boolean variables. Based on our observations, we believe that because JavaScript allows property type changes and property deletions at runtime, developers choose different coding idioms to implement a property needed only for the part of the program execution. Given our observations, it is difficult to summarize a common practice for coding this usage of JavaScript objects.

6. CONCLUSIONS AND FUTURE WORK

JavaScript is widely used for developing client-side web applications. Its dynamic characteristics (e.g., dynamic typing) pose software engineering challenges such as program understanding and security. In this paper, we performed a study on JavaScript websites to better understand object behavior. We defined and evaluated several dynamic metrics on certain categories of JavaScript objects (i.e., user objects and native objects). The metrics cover measurements of operations, object sizes, property changes, constructors, and prototype-based inheritance. From these results, we made several interesting observations:

- **The behavior of user objects is very different from objects in other categories.** User objects created by constructors actively use prototype-based inheritance. Their properties change often; for example, the local size of a user object is likely to expand after it is constructed. User objects are also likely to be associated with more operations than other categories of objects. Thus, a practical software tool for JavaScript requires a more accurate model for user objects.
- **JavaScript inheritance is not well understood.** The results on the use of a prototype-based inheritance do not exemplify good practices of object inheritance. (i) Immediate prototype objects are not reused often. (ii) Although many properties may be accessed via prototype objects, user objects rarely do this. Because the behavior of prototype-based inheritance is very different from class-based inheritance, JavaScript analysis techniques should build specific models for prototyping
- **Only a few native objects may affect program behavior.** Although there are many native objects in JavaScript applications, most of them are inactive in application code. Read operations on the local properties are the most frequently occurring operations on a native object.

DOI: 10.1002/spe

According to the observations, it makes sense to model JavaScript native objects based on their uses (e.g., [6]).

We investigated frequently occurring behavioral patterns for user objects including patterns of operations and property type changes.

- **Some operation occurrence patterns of user objects are widely observed across different websites.** Interesting operation occurrence patterns (e.g., pattern 4 in Table II) frequently occur in many websites, suggesting a common coding or user interaction style of JavaScript objects. Knowing those patterns may help improve understanding of JavaScript application behavior.
- **Certain websites exhibit more active property type changes.** Developers of some websites seems to intentionally implement specific property type change patterns more often. Each property type change pattern was found more frequently in a small number of websites. There were different dominant websites for each pattern.

Based on these findings and observations, we believe that specialized techniques should be used to analyze specific JavaScript objects because of their complicated behaviors.

- A flow-sensitive analysis (e.g., [21, 22, 34]) may accurately model the behavior of a JavaScript object whose properties frequently change at different program points.
- Because there are only a few immediate prototype objects whose properties are often inherited, the efforts to accurately model prototype-based inheritance (e.g., [9, 21]) should be focused on these prototype objects.
- Because there is a small fraction of native objects that may affect program behavior, specialized techniques (e.g., [6]) should be applied only for these native objects.

To the best of our knowledge, despite the fact that many of these techniques were present in the literature, there is no existing JavaScript static analyzer that can accurately analyze real JavaScript websites because of lack of scalability. We need to choose analysis techniques carefully for specific objects to achieve a sweet spot between precision and scalability for JavaScript analysis.

Because web technologies are developing quickly, in future work, we would like to examine how JavaScript web applications evolve and how evolution impacts object behavior. We also are interested in investigating more deeply into actual JavaScript code to observe the patterns identified, hoping to better summarize coding style. Finally, we would like to compare the behavior of JavaScript applications with software written in other dynamic programming languages (e.g., Ruby and Python).

REFERENCES

1. W3techs web technology surveys. Available from: http://w3techs.com/technologies/overview/client_side_language/all [last accessed May 2015].
2. Stackoverflow 2015 developer survey. Available from: http://stackoverflow.com/research/developer-survey-2015#tech [last accessed May 2015].
3. Lieberman H. Using prototypical objects to implement shared behavior in object-oriented systems. *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications, OOPLSA '86.* ACM: New York, NY, USA, 1986; 214–223.
4. Wegner P. Dimensions of object-based language design. *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications, OOPSLA '87.* ACM: New York, NY, USA, 1987; 168–182.
5. Schäfer M, Sridharan M, Dolby J, Tip F. Effective smart completion for JavaScript. *Technical Report RC25359*, IBM, 2013.
6. Madsen M, Livshits B, Fanning M. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013.* ACM: New York, NY, USA, 2013; 499–509.
7. Alimadadi S, Sequeira S, Mesbah A, Pattabiraman K. Understanding JavaScript event-based interactions. *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014.* ACM: New York, NY, USA, 2014; 367–377.
8. Guarnieri S, Livshits B. Gatekeeper: mostly static enforcement of security and reliability policies for JavaScript code. *Proceedings of the 18th Conference on USENIX Security Symposium, SSYM'09.* USENIX Association: Berkeley, CA, USA, 2009; 151–168.

DOI: 10.1002/spe

9. Guarnieri S, Pistoia M, Tripp O, Dolby J, Teilhet S, Berg R. Saving the world wide web from vulnerable JavaScript. *Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11.* ACM: New York, NY, USA, 2011; 177–187.
10. Wei S, Ryder BG. Practical blended taint analysis for JavaScript. *Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013.* ACM: New York, NY, USA, 2013; 336–346.
11. Gal A, Eich B, Shaver M, Anderson D, Mandelin D, Haghighat MR, Kaplan B, Hoare G, Zbarsky B, Orendorff J, Ruderman J, Smith EW, Reitmaier R, Bebenita M, Chang M, Franz M. Trace-based just-in-time type specialization for dynamic languages. *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09.* ACM: New York, NY, USA, 2009; 465–478.
12. Hackett B, Guo S-Y. Fast and precise hybrid type inference for JavaScript. *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12. ACM: New York, NY, USA, 2012; 239–250.
13. Richards G, Lebresne S, Burg B, Vitek J. An analysis of the dynamic behavior of JavaScript programs. *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10.* ACM: New York, NY, USA, 2010; 1–12.
14. Richards G, Hammer C, Burg B, Vitek J. The eval that men do: a large-scale study of the use of eval in JavaScript applications. *Proceedings of the 25th European Conference on Object-oriented Programming, ECOOP'11.* Springer-Verlag: Berlin, Heidelberg, 2011; 52–78.
15. Ratanaworabhan P, Livshits B, Zorn BG. JSMeter: comparing the behavior of JavaScript benchmarks with real web applications. *Proceedings of the 2010 USENIX Conference on Web Application Development, WebApps'10.* USENIX Association: Berkeley, CA, USA, 2010; 3–3.
16. Martinsen JK, Grahn H. A methodology for evaluating JavaScript execution behavior in interactive web applications. *Proceedings of the 2011 9th IEEE/ACS International Conference on Computer Systems and Applications, AICCSA '11.* IEEE Computer Society:, Washington, DC, USA, 2011; 241–248.
17. Ocariza Jr. FS, Pattabiraman K, Zorn B. JavaScript errors in the wild: an empirical study. *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering, ISSRE '11.* IEEE Computer Society: Washington, DC, USA, 2011; 100–109.
18. Ocariza F, Bajaj K, Pattabiraman K, Mesbah A. An empirical study of client-side JavaScript bugs. *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement* 2013; **0**:55–64.
19. Yue C, Wang H. Characterizing insecure JavaScript practices on the web. *Proceedings of the 18th International Conference on World Wide Web, WWW '09.* ACM: New York, NY, USA, 2009; 961–970.
20. Yue C, Wang H. A measurement study of insecure JavaScript practices on the web. *ACM Transaction on the Web* 2013; **7**(2):7:1–7:39.
21. Wei S, Ryder BG. State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. *Proceedings of the 28th European Conference on Object-oriented Programming.* Springer-Verlag: Berlin, Heidelberg, 2014; 1–26.
22. Jensen SH, Møller A, Thiemann P. Type analysis for JavaScript. *Proceedings of the 16th International Symposium on Static Analysis, SAS '09.* Springer-Verlag: Berlin, Heidelberg, 2009; 238–255.
23. Chugh R, Herman D, Jhala R. Dependent types for JavaScript. *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12.* ACM: New York, NY, USA, 2012; 587–606.
24. Lerner BS, Politz JG, Guha A, Krishnamurthi S. TeJaS: retrofitting type systems for JavaScript. *Proceedings of the 9th Symposium on Dynamic Languages, DLS '13.* ACM: New York, NY, USA, 2013; 1–16.
25. Schäfer M, Sridharan M, Dolby J, Tip F. Dynamic determinacy analysis. *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13.* ACM: New York, NY, USA, 2013; 165–174.
26. Andreasen E, Møller A. Determinacy in static analysis for jQuery. *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '14.* ACM: New York, NY, 2014; 17–31.
27. Dufour B, Driesen K, Hendren L, Verbrugge C. Dynamic metrics for Java. *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications, OOPSLA '03.* ACM: New York, NY, USA, 2003; 149–168.
28. Holkner A, Harland J. Evaluating the dynamic behaviour of Python applications. *Proceedings of the Thirty-second Australasian Conference on Computer Science - Volume 91, ACSC '09.* Australian Computer Society, Inc.: Darlinghurst, Australia, Australia, 2009; 19–28.
29. Furr M, An J-hD, Foster JS. Profile-guided static typing for dynamic scripting languages. *Proceedings of the 24th ACM Sigplan Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '09.* ACM: New York, NY, USA, 2009; 283–300.
30. Hills M, Klint P, Vinju J. An empirical study of PHP feature usage: a static analysis perspective. *Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013.* ACM: New York, NY, USA, 2013; 325–335.
31. Ryder BG. Dimensions of precision in reference analysis of object-oriented programming languages. *Proceedings of the 12th International Conference on Compiler Construction, CC'03.* Springer-Verlag: Berlin, Heidelberg, 2003; 126–137.

DOI: 10.1002/spe

32. Sridharan M, Dolby J, Chandra S, Schäfer M, Tip F. Correlation tracking for points-to analysis of JavaScript. *Proceedings of the 26th European Conference on Object-oriented Programming, ECOOP'12*. Springer-Verlag: Berlin, Heidelberg, 2012; 435–458.
33. Balakrishnan Gogul, Reps Thomas. Recency-abstraction for heap-allocated storage. *Proceedings of the 13th International Conference on Static Analysis, SAS'06*. Springer-Verlag: Berlin, Heidelberg, 2006; 221–239.
34. Kashyap V, Dewey K, Kuefner EA, Wagner J, Gibbons K, Sarracino J, Wiedermann B, Hardekopf B. JSAI: A static analysis platform for JavaScript. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*. ACM: New York, NY, USA, 2014; 121–132. http://doi.acm.org/10.1145/2635868.2635904.

DOI: 10.1002/spe