Analysis of JavaScript Programs: Challenges and Research Trends

KWANGWON SUN, Samsung Electronics, KAIST
SUKYOUNG RYU, KAIST

JavaScript has been a *de facto* standard language for client-side web programs, and now it is expanding its territory to general purpose programs. In this article, we classify the client-side JavaScript research for the last decade or so into six topics: static analysis, dynamic analysis, formalization and reasoning, type safety and JIT optimization, security for web applications, and empirical studies. Because the majority of the research has focused on static and dynamic analyses of JavaScript, we evaluate research trends in the analysis of JavaScript first and then the other topics. Finally, we discuss possible future research directions with open challenges.

Categories and Subject Descriptors: D.3.0 [**Programming Languages**]: General; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Language—*Program analysis*; D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Languages

Additional Key Words and Phrases: JavaScript, static analysis, dynamic analysis, analysis framework, web applications, security analysis

ACM Reference format:
Kwangwon Sun and Sukyoung Ryu. 2017. Analysis of JavaScript Programs: Challenges and Research Trends. *ACM Comput. Surv.* 50, 4, Article 59 (August 2017), 34 pages.
https://doi.org/10.1145/3106741

1 INTRODUCTION

Since the introduction of the JavaScript programming language in 1995, it has become the language for web programming and its applicability has been widely expanded to game engines (GitHub 2014; Stowasser 2015), compilers (Feldman 2015; Neighbors 2015), and even web operating systems (Atomic OS 2007; Evenrud 2014). JavaScript was originally developed as a simple scripting language, but it is now one of the most popular languages world-wide (TIOBE Software 2015; GitHub Blog 2015) thanks to its expressivity and portability. Because JavaScript supports dynamic language features, it can generate code at runtime, add object properties dynamically, and even delete them during program execution (Richards et al. 2010). In addition, JavaScript programs are often embedded in HTML documents in the form of web applications to handle user interactions,

This work is supported in part by Korea Ministry of Education, Science and Technology (MEST)/National Research Foundation of Korea (NRF) (Grant No. NRF-2014R1A2A2A01003235 and NRF-2017R1A2B3012020), Samsung Electronics, and Google.
Authors' address: K. Sun and S. Ryu, School of Computing, KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon, Republic of Korea 34141; emails: kwangwon.sun@samsung.com, sryu.cs@kaist.ac.kr.

https://doi.org/10.1145/3106741

and they are extremely portable because browsers provide JavaScript engines to run them (Lerner et al. 2013b).

While the expressivity and portability of JavaScript have brought its huge success, they also introduce various errors, vulnerabilities, and challenges in program analysis. Due to the excessively dynamic features and extensive use of implicit type conversion, JavaScript developers make type-related errors frequently (Pradel et al. 2015; Pradel and Sen 2015). Because most web applications contain untrusted third-party code, they are vulnerable to various security attacks (Yue and Wang 2009; Jang et al. 2010; Nikiforakis et al. 2012). Also, event-driven user interactions require understanding of complex semantics between JavaScript and HTML/DOM events (Alimadadi et al. 2014; Pradel et al. 2014). Researchers have proposed program analysis techniques (Jensen et al. 2009; Guarnieri et al. 2011; Sen et al. 2013; Alimadadi et al. 2014) to help JavaScript developers overcome such difficulties. However, since dynamic features and untrusted code are not available at compile time, and unrestricted event flows make modeling the execution behaviors for analysis an insurmountable difficulty, JavaScript analysis, especially static analysis, is extremely difficult.

In this article, we study the abundant client-side JavaScript research literature for the last 10 or so years. Starting from the JavaScript type system papers in 2005 (Thiemann 2005; Anderson et al. 2005), the JavaScript research community has grown hugely and rapidly; out of 31 papers presented at the 29th European Conference on Object-Oriented Programming (ECOOP'15), 10 papers (32%) discuss JavaScript issues. Not only academics but also industrial laboratories have been actively working on JavaScript (IBM 2007; Fournet et al. 2013; Google 2009; Bae et al. 2014), and research on other dynamic languages like PHP has been inspired by JavaScript research (Hills et al. 2013; Hauzar and Kofroň 2015). Our primary goal is to help researchers and developers grasp the big picture of the JavaScript research easily and quickly. To achieve the goal, we have collected 160 papers from the research community and classified them into six dominant research topics.[1] After introducing essential characteristics of JavaScript, its representative errors and vulnerabilities (Section 2), we evaluate research trends in the analysis of JavaScript first (Section 3) and then the other research topics (Section 4) because program analysis has been the most dominant topic in the JavaScript research. We discuss open challenges of JavaScript analysis and possible future research directions for them (Section 5), and we conclude (Section 6).

2 JAVASCRIPT CHARACTERISTICS

JavaScript was first developed as a simple scripting language in 1995, and its standardized specification, the ECMAScript specification, was first published in 1997. While the second and the third editions were released in 1998 and 1999, respectively, the proposed fourth edition was abandoned and it was only in 2009 when the fifth edition was finally published. In order to provide more error checking, the fifth edition introduced the "strict mode," which avoids error-prone language constructs. Starting from the sixth edition finalized in 2015, Ecma International plans to provide yearly updates to the ECMAScript specification.

In this section, we review JavaScript in terms of language characteristics, web environment characteristics, and JavaScript-specific errors and vulnerabilities as a background for understanding the rest of this article.

2.1 JavaScript Language Characteristics

JavaScript is a dynamically typed language and it supports extensive use of implicit type conversion. It provides seven kinds of types: `Undefined`, `Null`, `Boolean`, `Number`, `String`, `Symbol`, and `Object`. A JavaScript variable may have a value of any type during program execution, and a

[1] We explain our systematic methodology to identify and study dominant research topics in Appendix A.

```
1 get_cookie = function (name) {
2   var ca = document.cookie.split(';');
3   for (var i = 0, l = ca.length; i < l; i++) {
4     if (eval("ca[i].match(/\\b" + name + "=/)"))
5       return decodeURIComponent(ca[i].split('=')[1]);
6   }
7   return '';
8 }
```

Fig. 1. Dynamic code generation from `fiverr.com` (Jensen et al. 2012).

JavaScript value may be implicitly converted to a value of a different type according to the type conversion semantics defined in the language specification (ECMA 2015). Such features provide programmers with high expressivity and flexibility, but, at the same time, they can lead to type errors and unexpected behaviors. Flanagan (2006) summarized a set of implicit type conversions with peculiar behaviors like converting the empty string to `0` or `false`, and Pradel and Sen (2015) also reported an empirical study of unintuitive implicit type conversions in JavaScript.

JavaScript is a prototype-based object-oriented language. Unlike class-based object-oriented languages, an object may inherit properties of another object by setting it as a prototype object. An object may add and delete its properties dynamically. For example, suppose that an object `x` does not have a property named `p`. When a programmer attempts to assign a value `1` to the absent property `x.p`, statically typed languages like Java would throw an exception but JavaScript adds a new property `p` to `x` with an initial value `1`. Thus, objects can change their shapes during program execution, which is open to runtime-type errors and makes their static analysis extremely difficult.

In addition, JavaScript supports various dynamic language constructs. JavaScript programs can generate code to execute dynamically via the famous `eval` function and similar constructs including the `Function` constructor, `setInterval`, and `setTimeout`. JavaScript is statically scoped except for the `with` statement, which introduces a new scope at runtime. Also, object property names may be calculated dynamically; their values are not statically determined unlike field names and method names in Java, for example. All these dynamic features make static analysis of JavaScript programs challenging. Consider the example code from the web site `fiverr.com` (Jensen et al. 2012) in Figure 1. The conditional expression on line 4 calls the `eval` function to generate code from its string argument and to evaluate it. The string argument to the `eval` function is dynamically generated at every loop iteration, whose evaluation uses a different value for `ca[i]` for each loop iteration.

Finally, JavaScript is a functional language, which uses anonymous functions extensively. Until the sixth edition of the ECMAScript language specification (ECMA 2015) was released in June 2015, JavaScript did not provide any language-level module system and JavaScript developers used the module pattern (Miraglia 2007) with anonymous function calls. Many JavaScript libraries including jQuery (The jQuery Foundation 2006), the most popular JavaScript library with the market share of more than 90%,[2] use callback functions widely. Thus, even call graph construction of JavaScript programs is much more difficult than C and Java.

2.2 JavaScript Web Environment Characteristics

JavaScript programs execute within host environments, and web browsers are the most common host environment (Mozilla Developer Network 2005). JavaScript is often embedded within HTML

[2] http://w3techs.com/technologies/overview/javascript_library/all.

documents and web browsers provide host objects reflecting the HTML structure into JavaScript. In this article, we refer to HTML5 applications as *web applications*, which consist of three components: HTML for content structure, CSS for presentation, and JavaScript for user interaction. Web browsers provide the *Document Object Model (DOM)*, a tree representation of HTML documents, to enable JavaScript code to manipulate HTML.

Understanding behaviors of web applications requires analysis of interactions between DOM and JavaScript code and between JavaScript code and user inputs. DOM APIs provide interfaces for JavaScript code to query, traverse, and modify HTML contents, and JavaScript event handler functions allow users to interact with web applications. Web application developers may register event handlers statically using HTML attributes like `onload` and `onclick` or dynamically using `addEventListener`, and user-events can invoke them asynchronously. When an event handler is invoked, corresponding events may be propagated upward, downward, or both along DOM trees triggering more event handlers (W3C 2015). Such dynamic, asynchronous, and complex semantics of DOM APIs and event handlers make analysis of web applications onerous.

Unfortunately, because many web applications contain untrusted third-party code and dynamically loaded files, they are vulnerable to various security attacks (Yue and Wang 2009; Jang et al. 2010). Similarly, various browser extensions written in JavaScript, which have higher privileges than normal JavaScript code, make the problem more complex (Lerner et al. 2013b). Moreover, the Same-Origin Policy (SOP) adopted by most browsers is not secure enough. SOP allows contents from one "origin"[3] to read and modify other contents from the same origin only but not from other origins (Stuttard and Pinto 2011), which does not prohibit untrusted code from the same origin. Finally, various platforms for building server-side web applications in JavaScript, notably Node.js, introduce new challenges, which are beyond the scope of this article.

2.3 Type Errors and Security Vulnerabilities

JavaScript-specific characteristics introduce type errors and security vulnerabilities.

Type Errors. Literature (Anderson et al. 2005; Thiemann 2005) defines `TypeError` and `ReferenceError` as JavaScript type errors; `TypeError` denotes that the actual type of an operand is different from the expected type, and `ReferenceError` denotes that a name is used without any definition (ECMA 2015). For example, calling a non-function throws `TypeError` and reading an absent variable throws `ReferenceError`.

Security Vulnerabilities. Researchers have studied four kinds of security vulnerabilities in JavaScript (Guarnieri et al. 2011; Chugh et al. 2009; Bandhakavi et al. 2010; Tripp et al. 2014): *cross-site scripting (XSS)*, *open redirect*, *information leakage*, and *code injection*. First, web applications are vulnerable to client XSS[4] if an attacker can inject malicious HTML code into the values of parameters rendered by JavaScript code. Figure 2(a) from Tripp et al. (2014) illustrates an example client XSS. If the value of `URL` is `''http://...?val=<script> (malicious script) </script>''`, the code writes the malicious script to DOM. Second, open redirect denotes the case when JavaScript code redirects users to malicious sites. For example, when the value of `URL` is `''http://...?val=www.evil.com''` in Figure 2(a), the code redirects to the evil site. Third, information leakage denotes the case when sensitive information is leaked to outside. Figure 2(b) shows that cookie information may be leaked to outside over the network via `XMLHttpRequest`.

[3] Most browsers use the triple of protocol, host, and port as an origin.

[4] Among two kinds of XSS (server XSS and client XSS (OWASP 2013)), we focus on only client XSS in this article, because it is the only one relevant to client-side JavaScript.

```
var pos = document.URL.indexOf('val=')+4;
var val = document.URL.substring(pos,document.URL.length);
document.write(val);                              // client XSS
document.location.href = val;                     // open redirect
```

(a) Client XSS and open redirect [Tripp et al. 2014]

```
var req = new XMLHttpRequest();
req.open("POST", "https://www.remoteserver.com/");
req.send(document.cookie);                        // information leakage
eval(untrustedCode);                              // code injection
```

(b) Information leakage and code injection

Fig. 2. Four kinds of JavaScript security vulnerabilities.

Finally, untrusted code may be executed by dynamic code generation such as `eval` leading to code injection.

3 RESEARCH TRENDS IN ANALYSIS OF JAVASCRIPT

In this section, we evaluate research trends in program analysis of JavaScript. We classified the large body of the JavaScript research into six dominant research topics: static analysis, dynamic analysis, formalization and reasoning, type safety and JIT optimization, security for web applications, and empirical studies. The classification is based on the methodology described in Appendix A. Among six topics, static and dynamic analyses amount to about 57% of the papers we classified.

3.1 Static Analysis

Static analysis has been the most dominant research topic for client-side JavaScript applications. Out of 154 papers that we studied in this article,[5] 53 papers mainly discuss static analysis, which amount to about 34%. The extremely dynamic language features and web execution environments of JavaScript introduce various *challenges* in static analysis:

Dynamic Language Features

- *Dynamic Types.* Unlike statically typed languages like C and Java, JavaScript allows a variable to have values of different types at runtime, which hurts the static analysis precision.
- *Dynamic Constructs.* Various language constructs that generate code to execute dynamically makes it impossible to statically analyze the code precisely.
- *Dynamic Objects.* Similarly, dynamic addition and deletion of object properties, and dynamic calculation of object property names make precise static analysis of objects extremely difficult.

Web Execution Environments

- *DOM and Events.* Understanding behaviors of web applications needs analysis of interactions between DOM and JavaScript code and between JavaScript code and user inputs, which requires static analysis to address a vast amount of DOM APIs and to explore a huge space of user-driven event flows.
- *Analyzing Libraries.* Most web applications use large-scale libraries that contain complex code patterns and higher-order functions, which make static analysis unscalable.

[5] Among 160 papers we have collected, we focused on 154 papers relevant to our study.

In this section, we identify five research trends of JavaScript static analysis. To analyze JavaScript web applications in the wild, researchers have addressed challenging features gradually (Section 3.1.1), improved analysis precision by tackling stumbling blocks one by one (Section 3.1.2), improved analysis scalability by optimizing analysis techniques (Section 3.1.3), developed analysis frameworks for usability (Section 3.1.4), and applied static analysis to diverse areas (Section 3.1.5).

3.1.1 Extending Analysis Scope. Starting from small JavaScript subsets, JavaScript static analysis has extended to cover more features of JavaScript web applications.

JavaScript Subsets. Researchers have proposed various analyzers for their own statically analyzable subset languages. Guarnieri and Livshits (2009) proposed GATEKEEPER, which enforces security and reliability policies statically for a JavaScript subset called $\text{JavaScript}_{\text{SAFE}}$, and adds dynamic policy checks for another subset $\text{JavaScript}_{\text{GK}}$ that additionally supports non-static field stores and `innerHTML` assignments. Taly et al. (2011) defined SES_{light}, which supports a safe sandbox by making built-in objects immutable and supporting only restrictive forms of dynamic features like `eval`.

Dynamically Loaded Code. Because dynamically loaded code is not available at compile time, it is one of the main challenges for static analysis. However, JavaScript web applications often load external code dynamically: for example, `<script src = ‘‘http://source/dl.js’’>`. *Staged analysis* (Chugh et al. 2009; Guarnieri and Livshits 2010) addresses this problem by analyzing statically available code first on servers and then analyzing code when it is loaded dynamically on clients. Chugh et al. (2009) proposed a staged information flow analysis, which first inspects information flows of target applications without dynamically loaded code generating two kinds of residual checks for security policies, and then sends generated residual checks to clients to inspect dynamically loaded code. Guarnieri and Livshits (2010) proposed a staged pointer analysis, which analyzes points-to relationships of target applications on servers, and then sends them to clients via serialization to perform pointer analysis of dynamically loaded code.

Dynamic Features: `eval` *and* `with`. Notorious dynamic features of JavaScript that challenge static analysis are the `eval` function and the `with` statement. The `eval` function generates code at runtime, and the `with` statement is the only JavaScript construct that introduces scopes dynamically. To statically analyze such features as much as possible, researchers identified their usage patterns that can be rewritten without using the features. Jensen et al. (2012) developed *Unevalizer*, which statically tracks string values passed as arguments of the `eval` function, and rewrites the following three cases: (1) string constants, (2) JSON data, and (3) string values that can be estimated by specialized lattices or context-sensitive analysis. Richards et al. (2011b) identified common use patterns of `eval` that can be rewritten without using `eval`, and Meawad et al. (2012) developed `Evalorizer`, which executes target programs to collect string values passed as arguments of `eval`, and eliminates calls of `eval` that belong to the common patterns identified by Richards et al. (2011b). For the `with` statement, Park et al. (2013) investigated usage patterns of `with` and developed a rewriting mechanism: for variables within a `with` block, if they are properties of the `with` object, they are rewritten to explicit property accesses of the object. With this mechanism, they can rewrite all static occurrences of the `with` statement that do not include any dynamic code generation. Its implementation has been used in JavaScript analysis frameworks (Lee et al. 2012; Park et al. 2015a).

DOM and Events. Since most web applications embed JavaScript programs in HTML documents and run them via events, static analysis of web applications requires static analysis of DOM API calls and event calls. Statically analyzing event call flows is extremely difficult because most events are invoked by user inputs and they execute asynchronously. Jensen et al. (2011) represented DOM objects with corresponding abstract objects, and they analyzed events by considering all possible

combinations of event calls after analyzing top-level JavaScript code first. While their approach is sound, because it over-approximates DOM tree structure and event flows, the analysis results are also over-approximated. To improve the analysis precision, Tripp and Weisman (2011) and Tripp et al. (2014) adopted dynamic analysis, crawling, to collect DOM information. Instead of explicit modeling of DOM APIs, their static analysis used concrete DOM values that were obtained during dynamic analysis. Thus, while the analysis provides precise results, it sacrifices soundness. Park et al. (2015b) developed a more faithful DOM modeling, which makes a single abstract object for each DOM node and captures DOM tree structure more precisely.

Analyzing Libraries. Most web applications use several libraries written in JavaScript like jQuery or even written in other programming languages like platform-specific APIs. Because analyzing JavaScript libraries is difficult due to their complex structures and reflection for object initialization, JavaScript analyzers have used simple modeling for them instead of analyzing them. For platform-specific APIs written in native code, modeling has been an obvious solution. However, the modeling approach is labor-intensive, tedious, and error-prone. There are many JavaScript libraries (DefinitelyTyped 2013), and new versions require changes in existing modeling. Thus, researchers have proposed various techniques to analyze libraries rather than modeling them.

To improve the analysis scalability of jQuery, Schäfer et al. (2013) proposed a dynamic *determinacy analysis*, which dynamically collects determinacy facts like variables and expressions that are always evaluated to the same values, and specializes programs using the facts to help static analysis. While executing target programs, the determinacy facts are propagated until they can be affected by nondeterministic values such as program inputs and random values.

On the contrary, Andreasen and Møller (2014) exploited determinacy facts statically. They identified abstract values that have single concrete values in a given context as determinacy facts, and used them to provide more precise contexts for static analysis. Focusing on analyzing the jQuery library, they applied the static determinacy analysis for the following three techniques: parameter sensitivity that considers parameter values at call sites, loop specialization that analyzes each iteration separately, and context-sensitive heap abstraction that analyzes selected allocation sites more precisely.

For libraries in native code, Madsen et al. (2013) developed a *use analysis*, which analyzes such libraries without their code. It collects information about returned objects from library functions and property accesses of the objects, creates stub objects from the collected information, and uses them in static analysis.

While the analyses of JavaScript libraries, both dynamic and static determinacy analyses, are designed to be sound, the analysis of libraries in native code is not. Because the use analysis is simply based on the observable behaviors of the library functions, it may miss the library functionalities that are not revealed by returned objects.

3.1.2 Improving Analysis Precision. Another static analysis trend is to improve the analysis precision by handling dynamic features and loops more elaborately. Several approaches have used both static and dynamic information to analyze various dynamic features. To analyze dynamic objects more precisely, researchers have studied specific usage patterns of objects and proposed different object abstraction mechanisms. Because commonly used JavaScript libraries like jQuery use loops intensively, analysis techniques specialized for loop handling have been proposed.

Dynamic Features. To address more dynamic features beyond `eval` and `with`, various hybrid approaches proposed to use dynamic information for static analysis.

First, Guarnieri and Livshits (2009) performed static analysis on subset languages of JavaScript, and inspected excluded language constructs at runtime via code instrumentation. Similarly, Vogt

et al. (2007) and Just et al. (2011) analyzed explicit information flows dynamically while statically analyzing implicit information flows.

Second, several approaches perform dynamic analysis first to collect runtime information, specialize target programs using collected dynamic information, and perform static analysis on the specialized programs. Schäfer et al. (2013) proposed a dynamic determinacy analysis that infers determinacy facts, uses them to specialize target programs, and performs static pointer analysis on the programs. Similarly, Tripp and others (2011, 2014) collected DOM and location information of target programs by crawling dynamically, and used them to specialize and analyze target programs to detect vulnerabilities like client-side XSS and open redirect described in Section 2. Their tool is integrated into the IBM AppScan Standard Edition product (IBM 2007).

Finally, Wei and Ryder (2013) presented JSBAF, which performs static analysis on a set of traces collected dynamically. During the first dynamic phase, testers execute target programs to collect runtime information of execution traces including dynamically loaded and generated code. For the second static phase, JSBAF maps collected information of traces to their corresponding program paths, and performs static analysis on the specialized paths. While improving analysis precision, the static analysis is inherently limited to only collected traces.

Dynamic Objects. Because statically analyzing dynamic objects is onerous due to dynamic addition and deletion of object properties and even dynamic computation of object property names, researchers have tried various approaches to understand dynamic objects. Starting from simple imprecise pointer analyses for JavaScript objects (Jang and Choe 2009; Guarnieri and Livshits 2009), specific usage patterns of objects that cause imprecision have been studied and different object abstraction mechanisms have been proposed to represent objects precisely.

Jensen et al. (2009) and Heidegger and Thiemann (2010b) utilized *recency abstraction* (Balakrishnan and Reps 2006) to analyze object initialization patterns precisely. Because one of the root causes of the analysis imprecision is *weak update*, which updates values of object properties to joins of new values with old values during the computation of abstract values in the analysis, recency abstraction distinguishes the most recently allocated objects from joined old objects. By performing weak updates on joined old objects and *strong updates*, which overwrite previous abstract values, on the most recently allocated objects, they could maintain precise analysis results for recently allocated object initialization patterns.

Sridharan et al. (2012) proposed a *correlation tracking* technique that identifies patterns of reading and writing identical object properties and analyzes them precisely. Consider the following code pattern, which various JavaScript libraries including jQuery use frequently:

```
for (v in o1) { // for each property v in the object o1
     t = o1[v]; // copy the value of o1[v] to t
     o2[v] = t; // copy the value of o1[v] to o2[v]
}
o2.x();         // call the function o2.x
```

Let us assume that the object `o1` has three properties `x`, `y`, and `z`, and their values are some functions `func1`, `func2`, and `func3`, respectively. For an empty object `o2`, the `for` loop copies all the properties and their values in `o1` to `o2`, and the code calls the function `func1` by `o2.x()`. Traditional static analysis techniques would estimate that the possible values of `v` and `t` are {`x`, `y`, `z`} and {`func1`, `func2`, `func3`}, respectively, which estimates the possible values of `o2.x` to be {`func1`, `func2`, `func3`} losing precision. To analyze such cases precisely, the technique keeps track of correlated dynamic property access patterns specially.

Wei and Ryder (2014) extended JSBAF to support a state-sensitive pointer analysis that analyzes dynamic status of JavaScript objects precisely. It tracks addition and deletion of object properties

using property changes as status changes of objects, and it distinguishes object abstraction by object status, and uses object status as a context. Despite the increase in abstract states of objects, the authors reported that JSBAF incurs small overhead and it provides better precision than the correlation tracking technique (Sridharan et al. 2012).

As a fundamental approach for dynamic objects, Cox et al. (2014) presented the HOO (Heap with Open Objects) abstraction that represents JavaScript objects and heap very precisely. Considering objects with dynamic addition and deletion of their properties as "open objects," HOO represents objects using a relational abstract domain that captures object properties by partitioning them into sets of properties with operational information on them. While HOO can represent complex object operations precisely, it would incur notable performance overhead in practice.

Loops. Similarly for static analysis of other programming languages like C and Java, loops are one of the constructs that degrade the analysis precision for JavaScript. Because numbers of loop iterations may not be available at compile time, most static analyses join analysis results of all loop iterations, which leads to imprecision. Moreover, since the jQuery library uses loops intensively, researchers have proposed techniques to analyze loops precisely by distinguishing each loop iteration.

Andreasen and Møller (2014) proposed a static determinacy analysis to analyze each loop iteration separately. To analyze jQuery precisely, they applied different techniques for different kinds of loops. They unrolled `for` loops, and distinguished each iteration by selected local variables including loop control variables. Regarding `for-in` loops, they analyzed each loop iteration with its corresponding property in the loop condition separately. With multiple heuristics, they could analyze several versions of jQuery.

Park and Ryu (2015) proposed a *Loop-Sensitive Analysis (LSA)*, which treats different kinds of loops uniformly. LSA is basically a context-sensitive analysis that analyzes differect contexts separately; $\langle i, j, k \rangle$-LSA distinguishes the maximum i-depth nested loops, j-length loop iterations, and k-length call strings, and it analyzes them precisely. Unlike unrolling every loop for the same fixed number of times, LSA analyzes different loops with different numbers of iterations computed during the analysis. They proved the soundness and precision theorems using the Coq proof assistant tool (Coq 2012), and they reported that LSA could analyze all 14 released versions of jQuery.

3.1.3 Improving Analysis Scalability. The more JavaScript static analysis extends its analysis scope and improves its analysis precision, the more severe its analysis scalability issue becomes. Researchers have proposed approaches to eliminate the amount of unnecessary analysis computation, to sacrifice the analysis soundness by using unsound call graphs, and to study techniques to improve the analysis performance.

Removing Unnecessary Computation. Jensen et al. (2010) proposed *lazy propagation*, which propagates only necessary values of callee functions for inter-procedural analysis. When it analyzes a function, it identifies unaccessed object properties in the function and uses *unknown* abstract values for them. When such object properties are accessed, it invokes a recovery procedure to get their values from the function's callsites and propagates them. This mechanism obviously reduces memory consumption because the values of object properties are passed on demand. While extra computation of recovery procedures introduces overhead, the authors reported that the performance benefit from eliminating unnecessary computation overcomes the overhead.

Madsen and Møller (2014) developed a sparse dataflow analysis for JavaScript. While conventional sparse analysis techniques for C programs (Oh et al. 2014; Hardekopf and Lin 2011) are staged in the sense that the main analysis uses def-use flows built from a lightweight pre-analysis;

such a pre-analysis may not be lightweight for JavaScript due to its highly functional and dynamic features. Thus, the authors analyzed def-use flows on-the-fly during the main analysis.

Using Unsound Call Graphs. One notable approach is to sacrifice the analysis soundness to improve the analysis scalability. While losing soundness would lead to missing information about actual execution, some analysis clients like IDE supports may prefer scalability rather than soundness because IDEs should produce results to end-users in a few seconds while they can miss some suggestions.

For static analysis in such IDEs, Feldthaus et al. (2013) constructed JavaScript call graphs intentionally unsoundly and imprecisely. They made three decisions for scalability: (1) they use a field-based analysis, which uses a single abstract location for the same property name possibly in different objects, (2) they analyze only function objects ignoring other values, and (3) they ignore dynamic property accesses. While the constructed call graphs are obviously unsound and imprecise, they opened a new direction to pursue (Ko et al. 2015).

JSBAF (Wei and Ryder 2013) statically analyzes dynamically collected traces using unsound call graphs (Dufour et al. 2007). While it analyzes only a small subset of the actual execution traces, it can finish analysis of various benchmarks and web applications that pure static analyzers cannot finish. Similarly, Ko et al. (2015) formally presented a scalable framework that can statically analyze tunable subsets of target applications. One instance of their framework uses the unsound call graphs (Feldthaus et al. 2013).

Improving Analysis Performance. Dewey et al. (2015) developed a technique to parallelize static analyses. Unlike conventional approaches that parallelize existing sequential algorithms such as simultaneously analyzing nodes in worklist and branches in control flow graphs, their approach decomposes static analyses into two independent parts that can be parallelized: reachable-states computation and selectively merging multiple abstract states. They applied the proposed technique to JSAI, a JavaScript abstract interpreter, to analyze different function call contexts in parallel.

Recently, researchers have studied the relationships between the JavaScript analysis precision and scalability. While the research community on functional programming languages has known that k-CFA with a larger k is more precise but less scalable than that with a smaller k, it may not be true for JavaScript. Kashyap et al. (2014) reported that their experiments with JSAI showed that k-CFA with a larger k was often more scalable than that with a smaller k. In addition, analyses with correlation tracking (Sridharan et al. 2012), static determinacy (Andreasen and Møller 2014), loop sensitivity (Park and Ryu 2015), and more precise DOM modeling (Park et al. 2015b) showed that more context-sensitive analyses were more scalable than less sensitive analyses. JSAI, LSA, and the static determinacy analysis also support *heap cloning* (Lattner et al. 2007), which handles heap abstraction more precisely while providing better scalability.

3.1.4 Improving Analysis Framework Usability. Among static analysis frameworks developed for JavaScript programs, we discuss four open-source frameworks that have been actively maintained: TAJS, WALA, SAFE, and JSAI.

TAJS. Jensen et al. (2009) developed the Type Analyzer for JavaScript (TAJS), a JavaScript dataflow analysis that infers types and call graphs based on abstract interpretation, and they made it publicly available in 2012 (Møller et al. 2012). TAJS supports flow-sensitive and context-sensitive analyses, and it detects type-related errors such as invoking non-function values as functions and reading absent variables.

Starting from a simple type analysis (Jensen et al. 2009), TAJS extended its analysis techniques to analyze more JavaScript web applications. It uses *Unevalizer* (Jensen et al. 2012) to track string values passed to the `eval` function. It supports modeling of DOM and browser APIs, and it analyzes

events by considering all possible combinations of event calls (Jensen et al. 2011). TAJS can analyze several versions of jQuery (Andreasen and Møller 2014), and it provides various techniques like lazy propagation (Jensen et al. 2010) to improve the analysis scalability. Also, TAJS has been used in information flow analysis (Keil and Thiemann 2013).

WALA. The T. J. Watson Libraries for Analysis (WALA) (IBM Research 2006) has been successfully used for Java pointer analysis, and it was extended to support more languages including Android Java (Fuchs et al. 2009) and JavaScript. For JavaScript programs, WALA supports flow-insensitive and context-sensitive analyses. While it provides only sound pointer analysis for Java programs, it provides both sound propagation-based analysis and unsound field-based analysis for JavaScript. In addition to the dynamic determinacy analysis (Schäfer et al. 2013) and correlation tracking (Sridharan et al. 2012) for sound analysis of JavaScript, WALA also supports an unsound call graph construction (Feldthaus et al. 2013) for analysis scalability.

Various JavaScript studies have extended WALA for different purposes. Guarnieri et al. (2011) implemented security vulnerability detection via taint analysis on top of WALA, Tripp et al. (2014) extended WALA for hybrid security analysis, and WALA Delta[6] is a delta debugger for WALA-based JavaScript analyses. Recently, unsound call graphs of WALA have been used in a tunable sparse analysis (Ko et al. 2015).

SAFE. The Scalable Analysis Framework for ECMAScript (SAFE) (Lee et al. 2012) was designed to provide a general framework for multiple JavaScript analyzers. It supports flow-sensitive and context-sensitive analyses based on abstract interpretation. SAFE is open-sourced (KAIST PLRG 2012) with a formal specification of its internal representations and translations between them, which enabled a formal specification and reasoning of a JavaScript module system (Kang and Ryu 2012), and a set of desugaring rules for the `with` statement (Park et al. 2013).

SAFE has extended its target applications from pure JavaScript programs to web applications using various JavaScript libraries, DOM and browser APIs, and platform-specific APIs. Based on empirical studies about commonly used DOM and browser APIs, SAFE models a set of such APIs to analyze web applications precisely (Park et al. 2015b). For platform-specific APIs implemented in native code, it uses an automatic modeling mechanism and it can detect type-related errors including API misuses (Bae et al. 2014). SAFE is equipped with several techniques such as a loop-sensitive analysis (Park and Ryu 2015) and eliminating false positives from static analysis results using dynamic information (Park et al. 2016). Because SAFE was designed and developed in a pluggable way, it has been integrated with other tools like the Deckard code clone detector (Jiang et al. 2007) to empirically study JavaScript code clones (Cheung et al. 2015), and WALA to tune the analysis scalability (Ko et al. 2015).

JSAI. Kashyap et al. (2014) developed JSAI, a JavaScript abstract interpreter using an abstract machine-based semantics rather than a structural semantics as in TAJS and SAFE. JSAI provides high configurability to evaluate diverse analysis sensitivities, but it supports only limited modeling of built-in, DOM, and browser APIs. JSAI has been used in studies for a JavaScript type refinement (Kashyap et al. 2013), parallelization of abstract interpreters (Dewey et al. 2015), and security signature inference via information flow analysis (Kashyap and Hardekopf 2014).

3.1.5 Extending Analysis Clients. Taking advantage of JavaScript static analysis results, researchers have applied them to detect type-related errors, to detect security vulnerabilities, and to understand program behaviors.

[6] https://github.com/wala/WALADelta.

Type Error Detection. The most common use of JavaScript static analysis is to detect type-related errors. Starting from type errors in pure JavaScript benchmarks (Anderson et al. 2005; Thiemann 2005), researchers have studied type errors in web applications (Jensen et al. 2009; Park et al. 2015b), TypeScript declarations (Feldthaus and Møller 2014), and programs using MVC (Model-View-Controller) frameworks (Ocariza et al. 2015). Feldthaus and Møller (2014) developed TSCHECK, a tool that detects inconsistencies between TypeScript type declaration files and their corresponding JavaScript library implementation, which found 142 errors in the declaration files of 10 libraries, for example. Ocariza et al. (2015) developed AUREBESH, a tool that detects type inconsistencies in AngularJS (AngularJS 2010) applications, which is the most popular JavaScript MVC framework.[7] Because consistently using identifiers across associated models, views, and controllers is not trivial, and detecting such inconsistent uses is challenging as well, they inferred types of variables in models, and checked whether they are consistent in views and controllers.

Security Vulnerability Detection. A rich body of research has studied static analysis techniques for four kinds of JavaScript security vulnerabilities. *Cross-site scripting* can be detected by inspecting whether unsanitized URL strings are written to DOM via `document.write` or `innerHTML` (Guarnieri et al. 2011; Tripp et al. 2014). Similarly, *open redirect* can be prevented by detecting information flows where untrusted code changes the location information by overwriting the `document.location` property (Chugh et al. 2009; Guarnieri and Livshits 2009; Guarnieri et al. 2011). Researchers have detected *information leak* by inspecting flows from sensitive data like cookies and form inputs to outside over network or local files (Chugh et al. 2009). Finally, Guarnieri and Livshits (2009), Bandhakavi et al. (2010), and Guarnieri et al. (2011) detected *code injection* by inspecting whether untrusted scripts are passed to `eval`, `document.write`, and the `innerHTML` property.

Information flow analysis analyzes flows of information in programs, and it guarantees *confidentiality* and *integrity*. Confidentiality does not allow sensitive information flows to inappropriate destinations, and integrity restricts information flows from untrusted sources. An information flow may be *explicit* or *implicit*. An explicit flow occurs via a direct assignment of sensitive information to a vulnerable destination, and an implicit flow arises from control structures that utilize sensitive information (Hedin and Sabelfeld 2012). For example, if sensitive information is compared with a certain value in the `if` statement, then the result of comparison may disclose the sensitive information, breaking non-interference.

Static information flow analysis can track both explicit and implicit flows. Chugh et al. (2009) proposed a staged information flow analysis to handle dynamically loaded and generated JavaScript code. Bandhakavi et al. (2010) presented VEX, a framework for statically analyzing security vulnerabilities in browser extensions such as unsafe code injection and privilege escalation in Firefox extensions. They focused on `eval` and `innerHTML` for code injection and identified specific patterns of flows that break privilege levels in Firefox. Vogt et al. (2007) and Just et al. (2011) presented a hybrid analysis, which basically performs dynamic analysis and utilizes intra-procedural static analysis to capture implicit flows.

Researchers have also applied various different approaches. Guarnieri and Livshits presented tools like GATEKEEPER (2009) and GULFSTREAM (2010) that inspect security and reliability policies for web widgets. They defined security policies for a JavaScript subset language, and instrumented runtime checks for dynamic code generation. Taint analysis, an instance of information flow analysis, simply inspects whether tainted data in sources reach to sinks. Guarnieri et al. (2011) developed ACTARUS, a static taint analysis framework that detects code injection, XSS, and invalidated

[7] http://www.airpair.com/js/javascript-framework-comparison.

redirection. Karim et al. (2012) implemented Beacon, a static analyzer that detects capability leaks in the Jetpack framework (Mozilla 2014) of Mozilla. They modeled capabilities and accessing sensitive sources as tainted, and identified whether Jetpack modules and addons have capability leaks. Curtsinger et al. (2011) developed Zozzle that detects heap spraying attacks, which allocate many objects containing exploit code in victim applications' heap, using the dynamic heap-spraying detector Nozzle (Ratanaworabhan et al. 2009).

Program Understanding. Refactoring is one of the widely used techniques for enhancing program understanding. Unlike for statically typed languages like C and Java, refactoring for JavaScript is difficult due to the absence of type information. Feldthaus et al. (2011) developed JSRefactor, an automated refactoring framework for JavaScript, which uses pointer analysis results as static types of objects. The framework provides three kinds of refactoring: Rename changes property names of objects; Encapsulate Property sotres values of properties in local variables and provides functions to retrieve and modify them; and Extract Module creates a module with function closures. Because the pointer analysis was not scalable or applicable to incomplete programs like library code, Feldthaus and Møller (2013) developed LightRefactor, a semi-automatic Rename refactoring tool for JavaScript, which uses typing rules with lightweight static analysis. By restricting the functionality of the tool to be semi-automatic, focusing on only one kind of refactoring, and giving up soundness, LightRefactor was efficient enough to be usable in IDEs.

Code change impact analysis helps developers understand parts of a program that may be affected by a change in the program. Alimadadi et al. (2015) developed Tochal, a DOM-sensitive change impact analysis tool for JavaScript, which considers DOM elements, event functions, and asynchronous callbacks via XMLHttpRequest (XHR) objects. It builds static dependency graphs augmented with dynamic information for DOM-sensitive changes.

Finally, Pienaar and Hundt (2013) developed JSWhiz, a static analyzer that detects memory leaks in JavaScript web applications, as an extension to the open-source Closure compiler (Google 2009). The authors reported five common problem patterns for memory leaks and JSWhiz found new memory leaks from various Google products, but they are specific to the use of the Closure library and its event system abstractions.

3.2 Dynamic Analysis

Because of the extremely dynamic nature of JavaScript as we discussed in Section 2, dynamic analysis also has been an active research topic. We classify the literature on JavaScript dynamic analysis into three categories: JavaScript-specific testing techniques, crawling and dynamic symbolic execution for enlarging analysis coverage, and various applications of dynamic analysis like program understanding and error detection.

3.2.1 Testing Techniques. Several testing techniques have been proposed to address JavaScript-specific language features. First, because JavaScript code is often embedded in HTML documents, DOM manipulation is one of major issues to address. Mesbah and van Deursen (2009) and Pattabiraman and Zorn (2010) developed Atusa and DoDOM, respectively, that collect DOM invariants via dynamic analysis and use them as test oracles. Mirzaaghaei and Mesbah (2014) considered not only DOM structures but also their states to measure test coverage. They devised DOM-based coverage metrics and developed DomCovery, an open-source tool to measure the proposed coverage. In addition, Milani Fard et al. (2015) developed a technique that automatically generates DOM test fixtures by dynamic symbolic execution, and implemented it as a tool called ConFix.

Similarly for other dynamic languages like Scheme and Racket (Findler and Felleisen 2002), contracts (Meyer 1988) have been used for testing JavaScript programs. JSContest (Heidegger and Thiemann 2010a) supports random testing guided by type-like contracts that specify types

of inputs and outputs of functions. Researchers have extended contracts to specify access permissions (Heidegger et al. 2012) and to address higher-order behaviors of JavaScript via TreatJS (Keil and Thiemann 2015).

Artzi et al. (2011) developed Artemis, a framework for feedback-directed random testing (Pacheco et al. 2007). They considered event sequences as test inputs and exploited information of previous runs like read/write relations and branch coverage in event handlers to generate and prioritize next event sequences. Finally, Mirshokraie et al. developed MUTANDIS, a mutation testing tool with JavaScript-specific mutation operators (Mirshokraie et al. 2013a), and PYTHIA, a testing framework that automatically generates test oracles using mutation techniques (Mirshokraie et al. 2013b).

3.2.2 Crawling and Dynamic Symbolic Execution. Unlike sound static analysis that approximates all possible behaviors of programs, dynamic analysis is inherently unsound missing actual program executions. In order to enhance the dynamic analysis coverage, crawling and dynamic symbolic execution techniques have been proposed.

Crawling techniques collect DOM states by triggering events of web applications, and build state models by repeatedly triggering more events until no new DOM states are built. Researchers have studied ways to build state models that capture behaviors of target applications precisely and concisely. Mesbah et al. (2008, 2012) developed a tool called CRAWLJAX that builds state models concisely by merging similar states using the edit distance algorithm (Levenshtein 1996). Thummalapenta et al. (2013) crawled data using applications' GUI to generate state models for interesting behaviors and then to refine them using "business rules." In addition, Schur et al. (2013) developed PROCRAWL that crawls data from multiple users separately and generates an integrated state model.

Dynamic symbolic execution techniques symbolically execute programs with concrete values, collect path constraints on symbolic variables, and explore unvisited execution paths by controlling path constraints. Saxena et al. (2010) proposed a symbolic execution for JavaScript, Kudzu, which focused on precise handling of string values. They used dynamic symbolic execution for values and random exploration for event handlers. Maras et al. (2013) utilized dependencies between events to generate usage scenarios, and Li et al. (2014) developed SymJS, which enhanced event space exploration by using dynamic taint analysis for improving event sequence construction.

3.2.3 Program Understanding and Error Detection. Taking advantage of dynamic analysis results, researchers have developed various tools to help understand program behaviors and to improve error detection.

For program understanding, dynamic analysis has been used for execution trace analysis, fault localization, and IDE supports. Maras et al. (2012) presented a semi-automatic technique to extract JavaScript code responsible for certain behaviors by analyzing execution of usage scenarios. Alimadadi et al. (2014) developed a tool called CLEMATIS, which visualizes event traces in different levels by mapping between low-level event interactions and a high-level behavioral model. Jensen et al. (2015b) developed MEMINSIGHT, a memory profiler that traces time-varying memory behaviors and detects JavaScript-specific memory leaks, such as *closure-related drags* and *DOM leaks*. For fault localization, Ocariza Jr. et al. (2012) developed AUTOFLOX to localize JavaScript DOM-related errors automatically, and developed VEJOVIS (Ocariza Jr. et al. 2014) to suggest fixes to such faults. For IDE supports, Bajaj et al. (2014) developed DOMPLETION that provides code completion suggestions for JavaScript that interacts with DOM, and Hammoudi et al. (2015) applied the delta debugging technique (Cleve and Zeller 2000) to web applications.

Dynamic analysis techniques enabled detection of various errors including race conditions. Petrov et al. (2012) defined race vulnerabilities caused by asynchronous nature of web

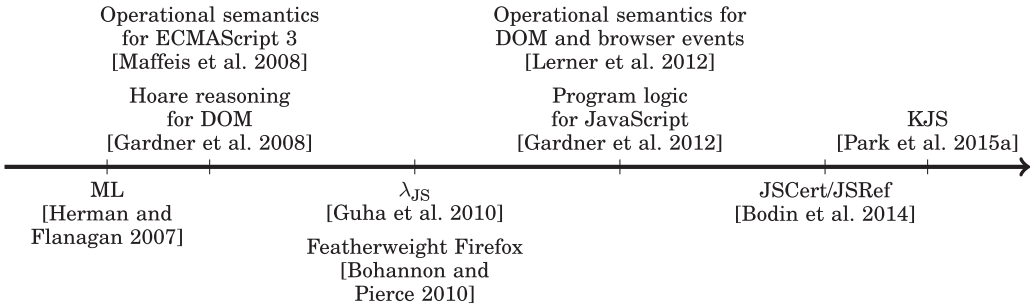

Fig. 3. Timeline of the formalism and reasoning topic.

applications, and implemented WebRacer, the first dynamic race detector for web applications. Raychev et al. (2013) defined race coverage and developed another dynamic race detector, EventRacer, which can detect more severe bugs than WebRacer with less false positives. Later, Jensen et al. (2015a) proposed a stateless model checker for event-driven applications, R4, which distinguishes harmful races from the races reported by EventRacer. In addition, EventBreak (Pradel et al. 2014) can detect event pairs that degrade performance, and DLint (Gong et al. 2015b) dynamically inspects code quality rules complementing existing lint-like static checkers.

Dynamic analysis has been also applied for information flow analysis. Sabre (Dhawan and Ganapathy 2009) tracks only explicit flows or only restricted forms of implicit flows, but often provides more precise results than static analysis techniques. Recently, Austin and Flanagan (2012) applied the secure multi-execution technique (Devriese and Piessens 2010), which executes programs multiple times with different security levels. The technique inherently guarantees non-interference because each execution is separated from one another. To lessen the performance overhead due to multiple executions, they introduced "faceted values" that contain raw values of different security levels, and execute programs once with faceted values.

Finally, Sen et al. (2013) developed Jalangi, a general dynamic analysis framework for JavaScript. The open-source tool provides simple dynamic analyzers, and it has been actively used in advanced dynamic analyses (Pradel et al. 2015; Pradel and Sen 2015; Gong et al. 2015a; Sen et al. 2015; Jensen et al. 2015b).

4 RESEARCH TRENDS OF OTHER TOPICS

In this section, we evaluate the trends of other research topics: formalization and reasoning, type safety and JIT optimization, security for web applications, and empirical studies.

4.1 Formalization and Reasoning

Research on formalization of the JavaScript language semantics mainly starts from Maffeis et al. (2008) and Guha et al. (2010), and it sprouts studies on JavaScript program reasoning and verification such as guaranteeing type safety and detecting security vulnerabilities as illustrated in Figure 3. Because the official JavaScript language standard, ECMAScript, informally describes the language semantics in prose, it is sometimes ambiguous and it contains bugs and infeasible behaviors (Park et al. 2015a). To provide a solid ground for JavaScript research, a correct and complete formal specification of the JavaScript semantics is necessary.

Maffeis et al. (2008) first proposed a small-step operational semantics for the full ECMAScript 3 specification. The operational semantics rules directly correspond to the description in the specification, and several studies (Maffeis and Taly 2009; Maffeis et al. 2009b) were built on top of

them. However, because they capture the quirky semantics of JavaScript too closely, they are quite lengthy, complex, and different from other language semantics based on the lambda calculus. In the course of developing the ECMAScript 4 specification, Herman and Flanagan (2007) defined an *executable* semantics of JavaScript by providing a "definitional interpreter" in ML. While it provides a runnable semantics, it is less comprehensible because implementation details obscure the core of the semantics.

As an alternative approach to formally describe the JavaScript semantics, Guha et al. (2010) developed λ_{JS}, a core calculus of the ECMAScript 3 semantics. They provided a "desugaring" process that rewrites most of the JavaScript concrete syntax into λ_{JS}, which is very similar to the standard lambda calculus. In addition, they made an interpreter implementation of λ_{JS} publicly available. Using the interpreter, they tested the λ_{JS} semantics and showed that the interpreter produces the same results as SpiderMonkey, V8, and Rhino implementations of JavaScript for a large set of the Mozilla JavaScript test suite. To show a possible application of λ_{JS}, they developed a type system that enforces a simple security property. Later, λ_{JS} was extended to address the ECMAScript 5 features like strict mode and accessors of objects (getters and setters) (Politz et al. 2012b). Thanks to its simplicity, λ_{JS} has been fairly used in formal reasoning of the JavaScript semantics (Politz et al. 2011, 2012a; Kang and Ryu 2012; Chugh et al. 2012a; Fournet et al. 2013).

Because JavaScript code often gets embedded and executed inside HTML documents as web applications, specification of JavaScript code behaviors requires specification of DOM and browsers. Gardner et al. (2008) formally specified the semantics of a small set of DOM manipulation, the W3C DOM Level 1 Core specification (W3C 1998), by using compositional Hoare logics. Bohannon and Pierce (2010) defined a small-step operational semantics of core functionalities of the Firefox web browser including event-driven behaviors and some simple JavaScript semantics. Lerner et al. (2012) rigorously specified behaviors of DOM events in the form of λ_{JS}-like operational semantics.

The aforementioned efforts to specify the JavaScript language semantics formally have been widely used in program reasoning and verification. Gardner et al. (2012) introduced a program logic adopted from the separation logic (Reynolds 2002) to track heap status during program evaluation. The logic can precisely capture non-trivial features of ECMAScript 3 such as prototype inheritance, restricted forms of the `eval` function, and the `with` statement. A number of studies use JavaScript formal semantics to guarantee type safety (Guha et al. 2011; Chugh et al. 2012a) and to prove security properties (Maffeis and Taly 2009; Maffeis et al. 2009b; Politz et al. 2011; Hedin and Sabelfeld 2012).

Recently, several approaches have applied to mechanize the JavaScript semantics. The JSCert project by Bodin et al. (2014) defined the semantics of a small subset of ECMAScript 5 using Coq and made the source code of a reference interpreter, JSRef, open to the public. They showed that JSRef is correct with respect to the mechanized semantics JSCert, and they also tested JSRef with the Test 262 test suite (ECMA TC39 2016). Park et al. (2015a) defined the entire semantics of ECMAScript 5, KJS, using the K framework (Rosu and Serbanuta 2014). Among the existing implementations of JavaScript, only KJS and Chrome V8's JavaScript implementation pass all 2,782 core language tests of Test 262. KJS has shown its effectiveness by revealing bugs in production JavaScript engines and finding security vulnerabilities.

4.2 Type Safety and JIT Optimization

While the JavaScript language itself does not provide any static type checking, researchers have proposed various type systems to guarantee diverse type safety properties and to improve the JIT compilation performance.

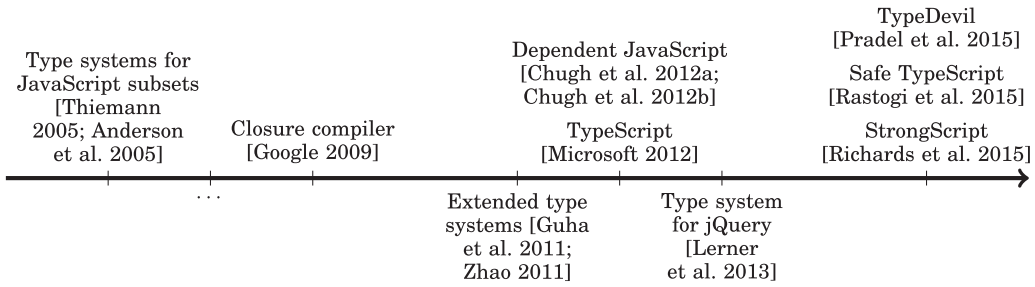

Fig. 4. Timeline of the type safety topic.

4.2.1 Type Safety. Research on various JavaScript type safety properties originates from the type systems of Thiemann (2005) and Anderson et al. (2005) as illustrated in Figure 4. Starting from them, various type systems have been proposed to address larger subsets of JavaScript and to detect more type-related errors more precisely. Recent approaches use not only static but also dynamic information like gradual type systems.

Thiemann (2005) devised a type system for a small subset of JavaScript, which detects "suspicious" type conversions like calling non-function objects, performing arithmetic operations on non-number objects, and accessing properties of `null` and `undefined` values. Anderson et al. (2005) developed a type system and a type inference algorithm for another small subset of JavaScript, and proved the type soundness. While they are the first attempts to provide type systems for JavaScript, their languages are unrealistically small.

Afterward, researchers have extended conventional type systems in various ways. Guha et al. (2011) proposed a flow-based type system, which utilizes flow analysis to reason about possible types of variables that are used in type testing patterns. Zhao (2011) defined a polymorphic type system with implicit extension of objects. Unlike the type system of Anderson et al. (2005) which supports only monomorphic types and explicit object extension, this work provides a polymorphic type inference algorithm to detect accesses to undefined members of objects. Chugh et al. (2012a, 2012b) applied SMT-based refinement logics to JavaScript types. They presented a core calculus, System D, in which all values have uniformly described nested refinement types, and they developed Dependent JavaScript (DJS) by extending it with strong updates to heap and heap unrolling. DJS can express runtime type tests, higher-order functions, extensible objects, prototype inheritance, and arrays.

Instead of detecting general type errors, type systems could be specialized in finding application-specific errors. Lerner et al. (2013) viewed the jQuery library (The jQuery Foundation 2006) as a query language for web applications' JavaScript code to interact with their enclosing Web page contents, and they defined a new kind of "jQuery errors." They identified query errors like returning too many, too few, or undesired DOM nodes as jQuery errors, because they return unexpected results from jQuery API calls. They developed a dependent type system to check whether jQuery APIs are used correctly. Because jQuery is the most widely used JavaScript library, analysis of jQuery uses has been one of the active research topics (Lerner et al. 2013; Andreasen and Møller 2014; Schäfer et al. 2013; Sridharan et al. 2012; Park and Ryu 2015). Finally, Pradel et al. (2015) focused on finding errors due to implicit type conversion. Based on the observation that uses of inconsistent types are often correlated with type errors even though implicit type conversion is allowed in the language semantics, they reported variables, properties, and functions that have multiple inconsistent types at runtime. They also proposed several techniques to reduce false positive reports.

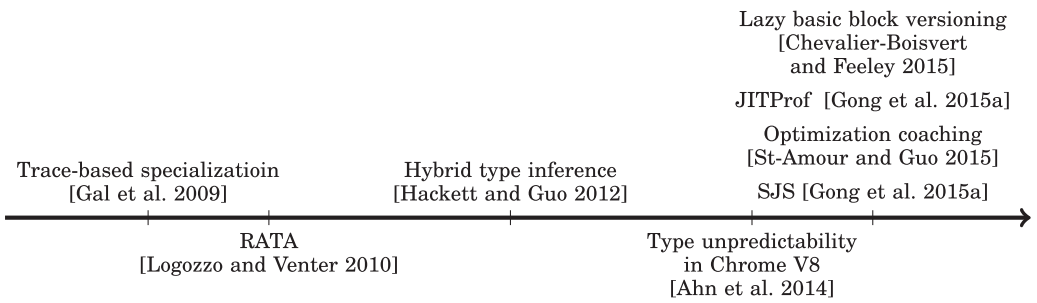

Fig. 5. Timeline of the JIT optimization topic.

The most widely used type systems for JavaScript in practice are based on *gradual typing*. Gradual type systems allow developers to use both static typing and dynamic typing in a single language by using type annotations. The Closure Compiler (Google 2009) compiles JavaScript code and generates an optimized and minimized version after checking its syntactic errors and simple type-related errors using users' type annotations. TypeScript (Microsoft 2012) is a superset of JavaScript from Microsoft, which offers a module system, classes, interfaces, and a gradual type system. Compilation of TypeScript code checks various type-related errors in terms of the extended language constructs, and generates a pure JavaScript code. To support backward compatibility with legacy JavaScript libraries, TypeScript declaration files for more than 300 JavaScript libraries are publicly available (DefinitelyTyped 2013).

While the type system of TypeScript is intentionally unsound, researchers have taken several approaches to make it safer. Rastogi et al. (2015) developed a "'Safe" compilation mode for TypeScript, which guarantees soundness by enforcing stricter static checks and runtime checks instrumented in compiled code. Richards et al. (2015) presented StrongScript, an extension of TypeScript with "concrete types" that are retained and used to optimize the program. Their implementation showed that StrongScript improves the runtime performance of typed programs on a modified version of the V8 JavaScript engine.

4.2.2 JIT Optimization. Modern compilers provide JIT optimization to improve runtime performance, but JIT optimization for JavaScript is often inefficient because JavaScript variables change their types frequently at runtime. While compilers for statically typed languages like Java can take advantage of compile-time type information to generate efficient code, JavaScript engines cannot rely on such information. To improve the performance of JIT optimized code, researchers have studied specializing runtime types by utilizing collected type information in various ways, and mitigating optimization failures with diverse techniques as summarized in Figure 5.

Gal et al. (2009) improved the JIT compilation performance using dynamic type information. They identified frequently executed traces at runtime, and generated specialized machine code for the dynamic types occurred on each trace. While the technique can generate optimized code for collected traces, it may not produce efficient code for the types not occurring on the collected traces. In contrast, Logozzo and Venter (2010) proposed RATA, a static analysis based on abstract interpretation for numeric type specialization. While numbers in JavaScript are the double-precision 64-bit floating point values as specified in the IEEE 754 standard, it statically identified variables that can be used as 32-bit integer types, and specialized them to have 32-bit integers during JIT compilation to improve performance. However, RATA involves a sophisticated mechanism to infer different numeric types precisely. Taking advantage of both static analysis and dynamic checks, Hackett and Guo (2012) achieved simplicity and better performance than RATA. During JIT compilation, when a statically inferred type is different from an observed type,

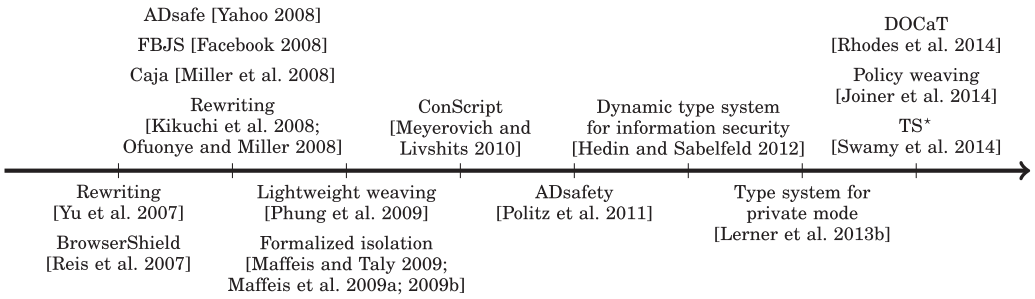

Fig. 6. Timeline of the security for web applications topic.

their system adds dynamic type updates. Afterward, static inference is augmented with such observed types at runtime.

Chevalier-Boisvert and Feeley (2015) proposed *lazy basic block versioning*, a JIT compilation technique that removes redundant dynamic type checks by using typing contexts recording type information of each program point. It creates different basic blocks for different typing contexts; when a new type is introduced during execution, the technique propagates the type information and generates new basic blocks. While their implementation showed promising results, their evaluation was limited to JavaScript benchmarks.

Besides type specialization, various approaches studied dealing with optimization failures. Ahn et al. (2014) examined root causes of *type unpredictability* that causes performance degradation in the Chrome V8 JavaScript compiler. They identified that because inherited prototype objects and method bindings are part of type definitions, they make type specialization especially difficult. Removing such requirements from V8 improved runtime performance. Gong et al. (2015a) defined code patterns that hinder JIT optimization, and developed JITProf, a profiling framework to dynamically identify such patterns. St-Amour and Guo (2015) developed an *optimization coaching* technique for JavaScript, which reports optimization failures to programmers with concrete recommendations of program changes that may make failed optimizations successful.

As an alternative of JIT compilation, Choi et al. (2015) proposed a static type system for a JavaScript subset, SJS, which enables ahead-of-time compilation. Because JIT compilation is ineffective when properties are dynamically added to JavaScript objects (Gong et al. 2015a), SJS addresses this problem by ensuring fixed object layouts. Similarly, another JavaScript subset asm.js (Mozilla 2013) provides better performance than JavaScript. While JIT compilers have less predictable performance based on complicated heuristics, asm.js provides predictable performance by eliminating dynamic type guards, boxed values, and garbage collection.

4.3 Security for Web Applications

As we discussed in Section 2, web applications are highly vulnerable to security attacks because they often run with untrusted code and their execution environments do not have a solid security mechanism. While a considerable amount of work has studied web security in general, we focus on web security research for client-side JavaScript in this article. We categorize the literature into two topics according to their techniques: program analysis as we discussed in Sections 3.1.5 and 3.2.3, and language-based methods described in this section.

Language-based approaches prohibit security vulnerabilities by constraining usability of JavaScript web applications as illustrated in Figure 6. By restricting expressivity of web applications, they ensure preventing language-specific security vulnerabilities of their interest.

4.3.1 Language Subsets. Subsetting approaches avoid security vulnerabilities by providing JavaScript subsets via forbidding vulnerable language constructs and syntactic usage. ADsafe (2008) of Yahoo restricts uses of security-vulnerable features such as global variables, the `this` variable, and the `eval` function. It also imposes syntactic restrictions that all accesses to DOM should be mediated by an ADSAFE object provided by a server. Similarly, FBJS (2008) of Facebook defines a JavaScript subset that isolates untrusted code via application-specific namespaces. Caja (Miller et al. 2008) of Google extends a JavaScript subset with features like freezing objects, protecting certain properties, and modularization to prevent object capability leaks (Noble et al. 2016).

4.3.2 Rewriting and Wrapping. Rewriting approaches transform programs by instrumenting them with runtime security checks, which does not require any browser modification. Yu et al. (2007) presented such an instrumentation as a set of formal rewriting rules, and Kikuchi et al. (2008) implemented it to support several web browsers. Similarly, BrowserShield (Reis et al. 2007) rewrites web pages including embedded JavaScript code to insert checks for vulnerability-driven filtering dynamically. Unfortunately, rewriting techniques require parsing and deep transformation at runtime, which incurs significant overhead. To reduce the runtime overhead of rewriting techniques, Ofuonye and Miller (2008) proposed to restrict the scope of instrumentation into the most vulnerable objects. In addition, Rhodes et al. (2014) developed DOCaT, a model checker that detects object capability leaks in an object capability system for JavaScript.

In order to avoid the runtime overhead of rewriting, wrapping approaches utilize lightweight techniques like *aspects* (Kiczales and Hilsdale 2001) to weave security policies into code. Phung et al. (2009) developed an aspect-oriented library, which allows JavaScript code to intercept particular method calls. While it does not cause much performance overhead, such shallow wrapping is insufficient for securing large applications. As a more general approach, Meyerovich and Livshits (2010) proposed ConScript built on top of Internet Explorer 8, which modified the JavaScript interpreter to weave even aliased functions.

As combined approaches, studies (Maffeis and Taly 2009; Maffeis et al. 2009a, 2009b) formalized subsetting, rewriting, and wrapping techniques using operational semantics to prove the validity of their mechanisms. They discuss how their methods can address the known vulnerabilities in FBJS and ADsafe. In addition, Joiner et al. (2014) utilized policy weaving that exploits static analysis to identify program points to be inspected, and inserts runtime checks to the points for monitoring given security policies.

4.3.3 Type Systems. Security type systems define specific security types and provide typing rules to enforce security policies as security types. Politz et al. (2011) defined accesses or assignments to security-vulnerable properties as type errors, and developed ADsafety, a type-based verification mechanism for ADsafe. Hedin and Sabelfeld (2012) devised a dynamic type system that guarantees non-interference, which prohibits information leaks from secret sources of programs to public sinks. Lerner et al. (2013b) proposed a type system that detects violations of the private browsing mode in web browsers, which are accesses from potentially unsafe code to provably safe code, from JavaScript extensions. Swamy et al. (2014) proposed TS*, a security type system for a gradually typed core of JavaScript. They introduced a new type, un, to denote values from untrusted JavaScript code and isolated code of type un from type-safe code.

4.4 Empirical Studies

To understand the behaviors of JavaScript programs in practice, researchers have performed various empirical studies as summarized in Figure 7. They mainly focused on dynamic features, security holes, and type-related errors.

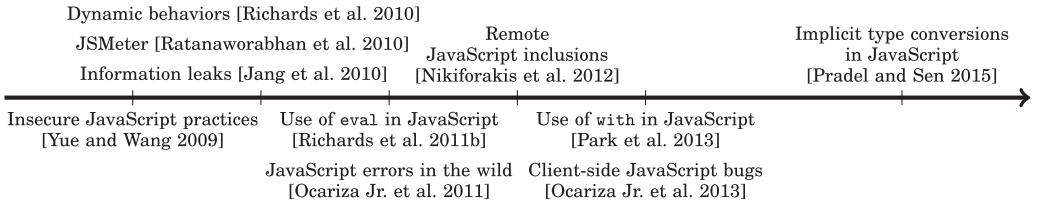

Fig. 7. Timeline of the empirical studies topic.

Richards et al. (2010) invalidated many common assumptions about the dynamic features of JavaScript from the literature based on their extensive investigation of JavaScript benchmarks and web sites with the most visitors. Their experimental results enabled several follow-up studies (Richards et al. 2011a, 2011b; Meawad et al. 2012). Similarly, JSMeter (Ratanaworabhan et al. 2010) showed that existing benchmarks do not reflect the behaviors of real-world web applications such as heavy use of events and short-lived functions. Richards et al. (2011b) and Park et al. (2013) studied real-world usage patterns of the `eval` function and the `with` statement, respectively, and they proposed methods to desugar such features. Wei et al. (2016) further investigated dynamic behaviors of JavaScript applications focusing on dynamic objects and tried to understand coding and user interaction practices.

Several studies investigated security vulnerabilities in real-world web applications. Yue and Wang (2009) studied insecure JavaScript practices such as using external scripts and dynamic code generation. They proposed to avoid using them or to use them in restricted ways like using external scripts in a sub-level HTML frame or an iframe document with a different origin. Jang et al. (2010) inspected numerous web sites using information flow analysis, and they identified four kinds of privacy-violating information flows: cookie stealing, location hijacking, history sniffing, and behavior tracking. Nikiforakis et al. (2012) identified four kinds of vulnerabilities due to remote code inclusions: cross-user and cross-network scripting, stale domain-name-based inclusions, stale IP-address-based inclusions, and typosquatting cross-site scripting.

Ocariza Jr. et al. (2011, 2013) studied root causes of bugs in web applications. They categorized most errors into four categories: permission denied, null exceptions, undefined symbols, and syntax errors. They thoroughly inspected 317 bugs from 12 bug repositories, and reported that 65% of the inspected bugs and 80% of the highest impact bugs are DOM-related. Recently, Pradel and Sen (2015) showed that most implicit type conversions are safe, and the most prevalent potentially harmful coercion is non-strict (in)equality checks for two objects of different types, which often leads to confusion.

5 OPEN CHALLENGES AND FUTURE RESEARCH DIRECTIONS

In this section, we discuss open challenges of JavaScript static analysis and its possible future research directions.

5.1 Open Challenges

Despite of a large amount of research effort in JavaScript static analysis over a decade, there still remain challenging tasks and new issues keep arising.

5.1.1 Libraries. As we discussed in Section 3.1.1, because most web applications use multiple libraries including JavaScript libraries and platform-specific native libraries, statically analyzing them is necessary for static analysis of web applications. The commonly used practices for statically analyzing such libraries are manually modeling their behaviors inside analyzers. While

manual modeling may provide precise semantics of even native libraries, it is labor-intensive, tedious, and error-prone. For JavaScript libraries, another option may be simply analyzing them as parts of target JavaScript applications. However, even the state-of-the-art JavaScript static analyzers are not yet able to analyze major libraries (Park and Ryu 2015).

One promising approach is to automatically model the semantics of libraries. Bae et al. (2014) automatically constructed modeling of platform-specific APIs from their specifications written in Web IDL.[8] Using such models, they could analyze interactions between JavaScript code and native library functions and detect possible misuses of library APIs. Similarly, Park (2014) automatically constructed modeling of JavaScript libraries from their specifications written in TypeScript. Compared to manual modeling, automatic modeling does not require much effort and it easily builds a model that captures a complete list of APIs. At the same time, purely automatic modeling may not be able to provide precise semantics of libraries.

5.1.2 Events. While web applications are mostly driven by events, statically analyzing event function calls is substantially difficult because most events are invoked by user interactions and they are executed asynchronously. Existing event analysis techniques (Jensen et al. 2011; Park et al. 2015b) soundly analyze all possible combinations of event function flows, which is slow and imprecise in analyzing web applications in the wild. Madsen et al. (2015) devised the event-based call graph, which can represent event-based flows precisely, and they showed that analyses using the graph can detect simple event-related bugs.

The problem with the current techniques is an enormous amount of event flows to analyze. Because the event space to investigate is infinite by nature, dynamically analyzing them piecemeal may be a reasonable option. As with existing approaches (Artzi et al. 2011; Li et al. 2014), analyzing finite event sequences generated in various ways can provide incomplete but usable analysis results. Similarly, analyzing subsets of the entire event space incrementally (Ko et al. 2015) can help understand event flows increasingly.

5.1.3 Iframes. IFrames, or inline frames, enable programmers to load separate HTML documents into an existing document, but no existing work supports static analysis of iframes. To analyze programs using iframes, static analyzers should analyze interactions between documents containing iframes and documents inside iframes. In addition, while the Same Origin Policy of JavaScript prohibits accesses from documents on different origins, the `postMessage` method allows communication between documents on different origins. Statically analyzing web applications using iframes and `postMessage` would be invaluable to detect security vulnerabilities.

5.1.4 Frameworks. Now that JavaScript is being used widely, various frameworks have been proposed to enhance productivity and performance. First, MVC frameworks help developers separate concerns by distinguishing three components of web applications: models, views, and controllers. Such components interoperate with each other according to their semantics. For example, in the AngularJS framework, models and views are two-way bound, and controllers require dependency injection to use functionalities of other controllers. In order to analyze applications using MVC frameworks statically, we should understand their interactions. Secondly, hybrid mobile frameworks allow users to write applications in both JavaScript and native code, which enables them to enjoy both the portability of web applications and device-specific functionalities of native applications. While hybrid applications are becoming popular, only a small number of papers address static analysis of hybrid applications (Lee et al. 2016; Brucker and Herzberg 2016).

[8] http://www.w3.org/TR/WebIDL/.

Statically analyzing hybrid applications may involve similar difficulties in static analysis of Java Native Interface (JNI), and existing research results on foreign function interfaces may be useful.

5.2 Promising Future Research Directions

Finally, based on our extensive studies, we discuss several promising research directions for JavaScript static analysis.

5.2.1 Soundiness. Recently, researchers have proposed "intentionally unsound" static analysis techniques. As a new term *soundiness* (Livshits et al. 2015) proposes:

> attempting to capture the balance, prevalent in practice, of over-approximated handling of most language features, yet deliberately under-approximated handling of a feature subset well recognized by experts

may be a reasonable approach to take. Given that statically analyzing large JavaScript web applications in a sound, scalable, and precise way is still an open problem, developers may get more benefits from analyzing subsets of program executions. Such soundy analyses may be sufficient for many analysis applications like IDEs, security analyses, and bug detectors that do not require sound analyses. Thus, Feldthaus et al. (2013) developed a mechanism to construct unsound call graphs, which were used in soundy analyses (Ko et al. 2015). Ko et al. (2015) formally presented a framework based on abstract interpretation, which tunes sound static analyses for better scalability using pre-analyses of selected program executions. Their implementation showed that using an unsound field-based analysis provided by WALA as a pre-analysis and a sound analysis provided by SAFE as a main analysis could analyze more programs than purely sound analyses.

5.2.2 Machine Learning. Inspired by active research results from the machine-learning community, program analyses have adopted machine-learning techniques by building statistical models from large codebases and predicting program properties using the models. JSNice (Raychev et al. 2015) predicts type annotations of untyped programs and identifier names of minified programs by using machine learning. Using a probabilistic model built from more than 10,000 JavaScript GitHub projects, it predicts code information by comparing similarities of code structures. Wei and Ryder (2015) presented an adaptive analysis, which selects a context-sensitive analysis from multiple options for each function. It first extracts characteristics of functions to predict appropriate sensitivity options for the functions, and then performs a main analysis using possibly different context-sensitivities for different functions. For choosing the "best" context-sensitivity option for a function, the authors applied machine-learning algorithms on a set of training programs using the list of function characteristics and corresponding analysis choices. While it is still in an early stage, devising analysis-specific machine-learning algorithms may be worthwhile.

5.2.3 Program Synthesis. Another active research area in programming languages is program synthesis, which has been used for synthesizing probabilistic programs, parallel graph programs, machine code, and parsers (Gill 2015). Heule et al. (2015) developed a method to automatically model libraries using search-based program synthesis. The method executes a target program while recording every memory access, and it generates code that models collected execution behaviors. Starting from initial models, the method incrementally adjusts them by random execution similar to Markov Chain Monte Carlo techniques. Utilizing such automatically generated library models, JavaScript static analyzers may be able to analyze large web applications.

5.2.4 Static Analysis of Programs in Other Programming Languages. Finally, recent approaches in the program analysis community for other programming languages may be applicable to

JavaScript static analysis. Yamaguchi et al. (2014) developed Joern, a code analysis platform for C and C++ using code property graphs combining ASTs, CFGs, and Program Dependency Graphs (PDGs). Utilizing a popular graph database Neo4J (Neo Technology 2010), Joern supports lightweight static analyses via graph traversal algorithms and it can detect security vulnerabilities in C and C++ programs. The INFER static analyzer from Facebook (Calcagno et al. 2015) supports *compositional analysis* based on separation logic, and it verifies code properties of every code modification in Facebook's mobile applications. Compared to the whole-program analyses used in existing JavaScript analyzers, compositional analysis is fast, scalable, and easily parallelizable, but it should be able to construct function summaries efficiently. While statically analyzing JavaScript programs involves different challenges from program analysis of other programming languages, novel approaches in their recent research may shed light on JavaScript analysis.

6 CONCLUSION

In this article, we survey a large body of research on JavaScript programs for the last decade or so. Using a systematic method based on reference relationships, we classify the literature into six dominant topics:

- *Static analysis.* The major research topic has been statically analyzing JavaScript web applications. Extremely dynamic features and web execution environments of JavaScript introduce various challenges in static analysis. Researchers have addressed such challenges by analyzing more features like dynamic constructs, events, and libraries. They have developed various techniques to improve analysis precision and scalability. Several static analyzers for JavaScript programs have been devised and they can help developers by detecting type-related errors and security vulnerabilities. Program analysis techniques including information-flow analysis have been widely applied to address security vulnerabilities in web applications.
- *Dynamic analysis.* To lessen the imprecision and scalability issues of static analysis, dynamic analysis like testing, crawling, and dynamic symbolic execution have been developed.
- *Formalization and reasoning.* Formal specification of the JavaScript language semantics provides a firm ground for reasoning quirky JavaScript behaviors rigorously.
- *Type safety and JIT optimization.* Various static type systems have proposed to guarantee diverse type safety properties and to improve the performance of JIT optimized code.
- *Security for web applications.* Language-based approaches have also addressed security vulnerabilities by defining manageable subset languages, type systems, and dynamic checking.
- *Empirical studies.* Because JavaScript programs behave quite differently from traditional languages such as C and Java, researchers empirically studied dynamic features and vulnerabilities specific to JavaScript.

The JavaScript research community has made remarkable achievements so far. Various open-source tools are developed and being used to detect bugs in real-world web applications. Researchers helped Ecma International to add more error checking features to the ECMAScript specification. Developers in industry such as Facebook, Google, IBM, Oracle, and Samsung use tools to improve the JavaScript code quality.

At the same time, there still remain abundant open challenges: large-scale libraries, asynchronous event flows via user inputs, interaction between iframes, and new analysis domains like MVC frameworks and hybrid applications. Fortunately, recent ideas from other research areas such as machine learning and program synthesis have opened the gate to promising results.

APPENDIX

A METHODOLOGY FOR RESEARCH TREND ANALYSIS

In this Appendix, we explain how we analyzed client-side JavaScript research trends systematically. We first collected 160 JavaScript-related papers for the last 10 or so years from the research community. We assembled papers published in major programming languages and software engineering conferences,[9] and we collected papers from active JavaScript research groups. Then, we manually reviewed all the references of the collected papers to check any obviously missing papers. To analyze the assembled papers systematically and objectively, we used a *citation graph* (Section A.1) to identify dominant research topics and to evaluate trends of each research topic (Section A.2).

A.1 Citation Graph

In order to identify dominant research topics and research trends from the papers without much biased views, we used a citation graph (Lu et al. 2006). A citation graph is a directed graph with papers for nodes and citing relationships for edges. An edge from a node X to another node Y denotes that a paper X cites another paper Y. Thus, nodes in a citation graph are ordered from the past to the present. Figure 8(a) shows the citation graph built from the assembled JavaScript-related papers. A node with many in-edges may be more important than a node with a small number of in-edges, and a set of nodes connected may be more closely related than other nodes.

A.2 Research Trend Analysis Using Citation Graph

A.2.1 Analysis of Research Topics with Hub Graphs. Because the citation graph shown in Figure 8(a) is too complex to investigate, we constructed its subgraphs called *hub graphs* to tune numbers of nodes to inspect. A hub graph with degree n consists of nodes that have more than or equal to n in-edges from the citation graph. By changing degrees of hub graphs, we could analyze 160 papers incrementally.

Using hub graphs, we identified dominant topics in JavaScript research. Figure 8(b) shows our initial hub graph with an arbitrary degree 20, where nodes are labeled by paper numbers summarized in a companion material (Sun and Ryu 2015). For each edge in the hub graph, we reviewed its corresponding reference, and we removed references that are not closely related to the cited paper. After reviewing all the edges, we identified groups of closely connected nodes as research topics. For the hub graph with degree 20, we identified four topics: "semantics ()," "security for web applications ()," "static analysis ()," and "empirical studies ()."

Then, by relaxing degrees of hub graphs, we refined research topics gradually. Figure 8(c) illustrates a hub graph with degree 15, which has more nodes than the initial hub graph with degree 20 so that we can identify more research topics. This hub graph refines the previous "security for web applications" topic into "language-based security ()" and "program analysis for security ()"; and it also refines "static analysis" into "type system ()" and other "static analysis ()." Note that while some papers are relevant to multiple topics, we categorized papers disjointly by choosing more relevant topics to them. Finally, Figure 8(d) shows a hub graph with degree 10, which introduces two more topics: "JIT optimization ()" and "dynamic analysis ()." The nodes and edges of the hub graphs in Figure 8 are all available in the companion material (Sun and Ryu 2015).

After several rounds of constructing hub graphs with different degrees, we identified six research topics in JavaScript as summarized in Table 1. We ended up using the hub graph with degree 10 and Table 1 also shows the papers in the hub graph for each topic. In this article, we

[9] ASE, ECOOP, ESOP, FSE, ICSE, OOPSLA, PLDI, and POPL.

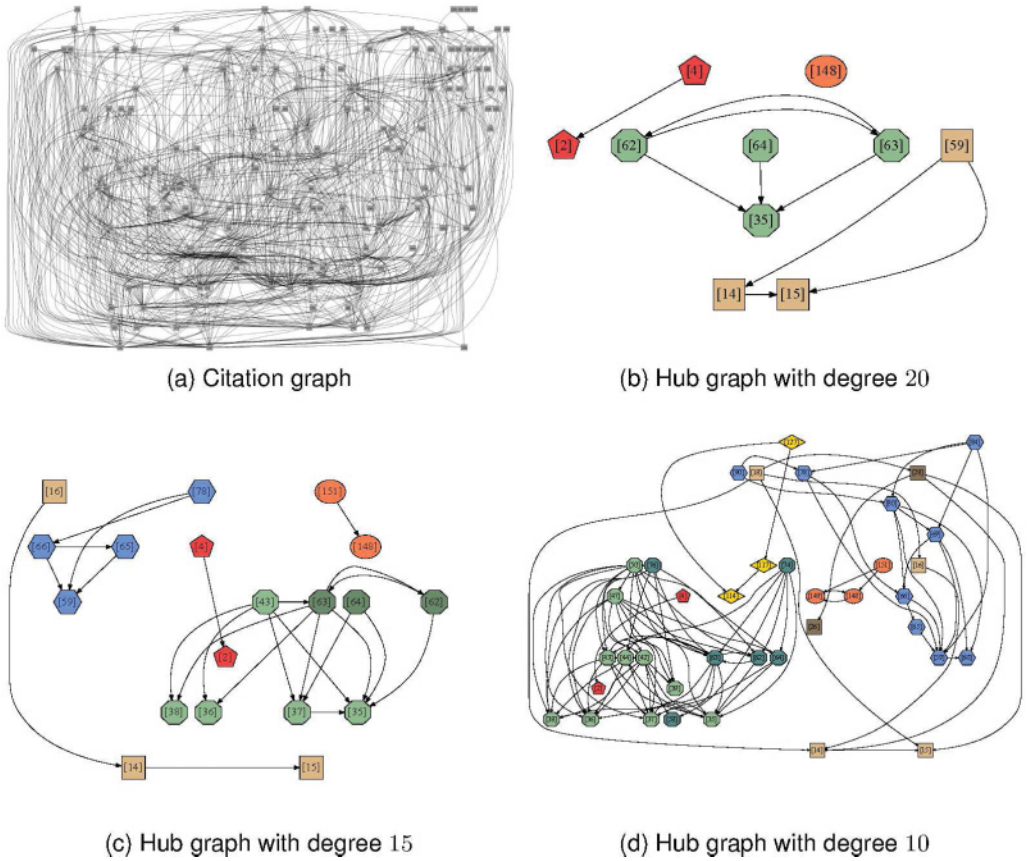

Fig. 8. Citation graph and hub graphs for JavaScript-related papers.

Table 1. JavaScript Research Topics and Well-Cited Papers

Research topic		Major studies
Formalization and reasoning		Maffeis et al. (2008) and Guha et al. (2010)
Type safety and JIT optimization		Anderson et al. (2005), Thiemann (2005), Gal et al. (2009), Chugh et al. (2012a), Guha et al. (2011), and Hackett and Guo (2012)
Security for web applications	Language-based methods	Yu et al. (2007), Reis et al. (2007), Yahoo (2008), Facebook (2008), Miller et al. (2008), Maffeis et al. (2009a), Phung et al. (2009), Maffeis and Taly (2009), Meyerovich and Livshits (2010), and Politz et al. (2011)
	Program analysis	Vogt et al. (2007), Chugh et al. (2009), Guarnieri and Livshits (2009), Taly et al. (2011), and Guarnieri et al. (2011)
Static analysis		Jensen et al. (2009), Jang and Choe (2009), Jensen et al. (2010), Heidegger and Thiemann (2010b), Jensen et al. (2011), Sridharan et al. (2012), Jensen et al. (2012), Madsen et al. (2013), and Wei and Ryder (2013)
Dynamic analysis		Saxena et al. (2010), Artzi et al. (2011), and Sen et al. (2013)
Empirical studies		Richards et al. (2010), Ratanaworabhan et al. (2010), and Richards et al. (2011b)

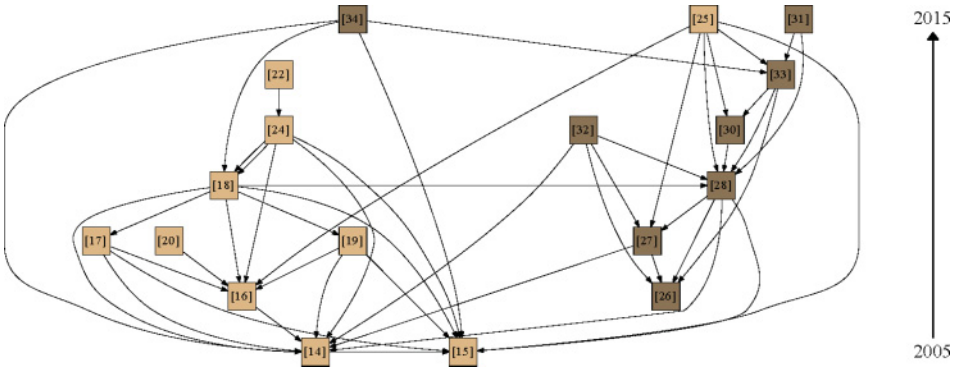

Fig. 9. Citation graph for type safety and JIT optimization.

discuss the "security for web applications using program analysis" topic as a part of the "static and dynamic analysis" topic (Sections 3.1.5 and 3.2.3).

A.2.2 Analysis of Trends in Research Topics with Citation Graph. Finally, we analyzed trends of each research topic using the corresponding subgraph of the full citation graph. By reviewing all the papers in each topic chronologically, we studied the research directions of past 10 or so years. For example, consider the citation graph for the "type safety and JIT optimization" topic illustrated in Figure 9. Research on the "type safety" topic started from two papers in 2005 ([14, 15] in Figure 9) (Anderson et al. 2005; Thiemann 2005) followed by various extensions ([16, 17]) (Guha et al. 2011; Zhao 2011) including refinement types ([18, 19]) (Chugh et al. 2012a, 2012b). Recent research results address implementation-specific type errors ([20]) (Lerner et al. 2013), dynamic type analysis ([25]) (Pradel et al. 2015), and unsoundness of TypeScript ([22, 24]) (Richards et al. 2015; Rastogi et al. 2015). As for the "JIT optimization" topic, earlier work focused on trace-based type specialization ([26, 27, 28]) (Gal et al. 2009; Logozzo and Venter 2010; Hackett and Guo 2012), but recent research efforts have spent on handling optimization failures ([30,33,31]) (Ahn et al. 2014; Gong et al. 2015a; St-Amour and Guo 2015), introducing a new specialization technique and proposing an alternative of JIT compilation ([32,34]) (Chevalier-Boisvert and Feeley 2015; Choi et al. 2015).

REFERENCES

Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas. 2014. Improving JavaScript performance by deconstructing the type system. In *Proc. of the Conference on Programming Language Design and Implementation*. 496–507.

Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2015. Hybrid DOM-sensitive change impact analysis for JavaScript. In *Proc. of the European Conference on Object-Oriented Programming*. 321–345.

Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2014. Understanding JavaScript event-based interactions. In *Proc. of the International Conference on Software Engineering*. 367–377.

Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards type inference for JavaScript. In *Proc. of the European Conference on Object-Oriented Programming*. 428–452.

Esben Andreasen and Anders Møller. 2014. Determinacy in static analysis for jquery. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 17–31.

AngularJS. 2010. AngularJS. Retrieved from http://www.angularjs.org.

Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A framework for automated testing of JavaScript web applications. In *Proc. of the International Conference on Software Engineering*. 571–580.

Atomic OS. 2007. Atomic OS. Retrieved from http://atomos.sourceforge.net/aos-1.1/index.html.

Thomas H. Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In *Proc. of the Symposium on Principles of Programming Languages*. 165–178.

SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. SAFEWAPI: Web API misuse detector for web applications. In *Proc. of the International Symposium on Foundations of Software Engineering*. 507–517.

Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2014. Dompletion: DOM-aware JavaScript code completion. In *Proc. of the International Conference on Automated Software Engineering*. 43–54.

Gogul Balakrishnan and Thomas Reps. 2006. Recency-Abstraction for heap-allocated storage. In *Proc. of the International Symposium on Static Analysis*. 221–239.

Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. 2010. VEX: Vetting browser extensions for security vulnerabilities. In *Proc. of the USENIX Conference on Security*. 22–22.

Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A trusted mechanised JavaScript specification. In *Proc. of the Symposium on Principles of Programming Languages*. 87–100.

Aaron Bohannon and Benjamin C. Pierce. 2010. Featherweight firefox: Formalizing the core of a web browser. In *Proc. of the USENIX Conference on Web Application Development*. 11–11.

Achim D. Brucker and Michael Herzberg. 2016. On the static analysis of hybrid mobile apps. In *Proc. of the International Symposium on Engineering Secure Software and Systems*. 72–88.

Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter OHearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *Proc. of NASA Formal Methods*. 3–11.

WaiTing Cheung, Sukyoung Ryu, and Sunghun Kim. 2015. Development nature matters: An empirical study of code clones in JavaScript applications. *Empirical Software Engineering* (2015), 1–48.

Maxime Chevalier-Boisvert and Marc Feeley. 2015. Simple and effective type check removal through lazy basic block versioning. In *Proc. of the European Conference on Object-Oriented Programming*. 101–123.

Wontae Choi, Satish Chandra, George C. Necula, and Koushik Sen. 2015. SJS: A type system for JavaScript with fixed object layout. In *Proc. of the International Symposium on Static Analysis*. 181–198.

Ravi Chugh, David Herman, and Ranjit Jhala. 2012a. Dependent types for JavaScript. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 587–606.

Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. 2012b. Nested refinements: A logic for duck typing. In *Proc. of the Symposium on Principles of Programming Languages*. 231–244.

Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. 2009. Staged information flow for JavaScript. In *Proc. of the Conference on Programming Language Design and Implementation*. 50–62.

Holger Cleve and Andreas Zeller. 2000. Finding failure causes through automated testing. In *Proc. of the International Workshop on Automated Debugging*.

Coq. 2012. The Coq Proof Assistant. Retrieved from http://coq.inria.fr/.

Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. 2014. Automatic analysis of open objects in dynamic language programs. In *Proc. of the International Symposium on Static Analysis*. 134–150.

Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2011. ZOZZLE: Fast and precise in-browser JavaScript malware detection. In *Proc. of the USENIX Conference on Security*. 3–3.

DefinitelyTyped. 2013. DefinitelyTyped. Retrieved from http://definitelytyped.org.

Dominique Devriese and Frank Piessens. 2010. Noninterference through secure multi-execution. In *Proc. of the Symposium on Security and Privacy*. 109–124.

Kyle Dewey, Vineeth Kashyap, and Ben Hardekopf. 2015. A parallel abstract interpreter for JavaScript. In *Proc. of the International Symposium on Code Generation and Optimization*. 34–45.

Mohan Dhawan and Vinod Ganapathy. 2009. Analyzing information flow in JavaScript-based browser extensions. In *Proc. of the Annual Computer Security Applications Conference*. 382–391.

Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2007. Blended analysis for performance understanding of framework-based applications. In *Proc. of the International Symposium on Software Testing and Analysis*. 118–128.

ECMA. 2015. ECMA-262: ECMAScript 2015 Language Specification (6th ed). (June 2015). http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

ECMA TC39. 2016. ECMAScript Language test262. Retrieved from https://github.com/tc39/test262.

Anders Evenrud. 2014. OS.js: JavaScript cloud/web desktop platform. Retrieved from http://os.js.org.

Facebook. 2008. FBJS: FaceBook JavaScript. Retrieved from https://github.com/facebook/fbjs.

Carter Feldman. 2015. CCJS v1.2: The world's first C compiler written in JavaScript. Retrieved from http://carterf.com/cf.gy/ccjsPublic/.

Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. 2011. Tool-supported refactoring for JavaScript. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 119–138.

Asger Feldthaus and Anders Møller. 2013. Semi-automatic rename refactoring for JavaScript. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 323–338.

Asger Feldthaus and Anders Møller. 2014. Checking correctness of typescript interfaces for JavaScript libraries. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 1–16.

Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proc. of the International Conference on Software Engineering*. 752–761.

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *Proc. of the International Conference on Functional Programming*. 48–59.

David Flanagan. 2006. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc.

Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. 2013. Fully abstract compilation to JavaScript. In *Proc. of the Symposium on Principles of Programming Languages*. 371–384.

Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. 2009. *SCanDroid: Automated Security Certification of Android Applications*. Technical Report CS-TR-4991. University of Maryland.

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proc. of the Conference on Programming Language Design and Implementation*. 465–478.

Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. 2012. Towards a program logic for JavaScript. In *Proc. of the Symposium on Principles of Programming Languages*. 31–44.

Philippa A. Gardner, Gareth D. Smith, Mark J. Wheelhouse, and Uri D. Zarfaty. 2008. Local Hoare reasoning about DOM. In *Proc. of the Symposium on Principles of Database Systems*. 261–270.

Andy Gill (Ed.). 2015. PLDI'15. *ACM SIGPLAN Notices* 50, 6 (2015).

GitHub. 2014. JavaScript game engines. Retrieved from https://github.com/showcases/javascript-game-engines.

GitHub Blog. 2015. Language Trends on GitHub. Retrieved from https://github.com/blog/2047-language-trends-on-github.

Liang Gong, Michael Pradel, and Koushik Sen. 2015a. JITProf: Pinpointing JIT-unfriendly JavaScript code. In *Proc. of the Joint Meeting on Foundations of Software Engineering*. 357–368.

Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015b. DLint: Dynamically checking bad coding practices in JavaScript. In *Proc. of the International Symposium on Software Testing and Analysis*. 94–105.

Google. 2009. Closure Compiler. Retrieved from https://developers.google.com/closure/compiler/.

Salvatore Guarnieri and Benjamin Livshits. 2009. GATEKEEPER: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proc. of the USENIX Conference on Security*. 151–168.

Salvatore Guarnieri and Benjamin Livshits. 2010. GULFSTREAM: Staged static analysis for streaming JavaScript applications. In *Proc. of the USENIX Conference on Web Application Development*. 6–6.

Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. 2011. Saving the world wide web from vulnerable JavaScript. In *Proc. of the International Symposium on Software Testing and Analysis*. 177–187.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The essence of JavaScript. In *Proc. of the European Conference on Object-Oriented Programming*. 126–150.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2011. Typing local control and state using flow analysis. In *Proc. of the European Symposium on Programming*. 256–275.

Brian Hackett and Shu-yu Guo. 2012. Fast and precise hybrid type inference for JavaScript. In *Proc. of the Conference on Programming Language Design and Implementation*. 239–250.

Mouna Hammoudi, Brian Burg, Gigon Bae, and Gregg Rothermel. 2015. On the use of delta debugging to reduce recordings and facilitate debugging of web applications. In *Proc. of the Joint Meeting on Foundations of Software Engineering*. 333–344.

Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization*. 289–298.

David Hauzar and Jan Kofroň. 2015. Framework for static analysis of PHP applications. In *Proc. of the European Conference on Object-Oriented Programming*. 689–711.

Daniel Hedin and Andrei Sabelfeld. 2012. Information-Flow security for a core of JavaScript. In *Proc. of the Computer Security Foundations Symposium*. 3–18.

Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. 2012. Access permission contracts for scripting languages. In *Proc. of the Symposium on Principles of Programming Languages*. 111–122.

Phillip Heidegger and Peter Thiemann. 2010a. Contract-driven testing of JavaScript code. In *Proc. of the International Conference on Objects, Models, Components, Patterns*. 154–172.

Phillip Heidegger and Peter Thiemann. 2010b. Recency types for analyzing scripting languages. In *Proc. of the European Conference on Object-Oriented Programming*. 200–224.

David Herman and Cormac Flanagan. 2007. Status report: Specifying JavaScript with ML. In *Proc. of the Workshop on ML*. 126–150.

Stefan Heule, Manu Sridharan, and Satish Chandra. 2015. Mimic: Computing models for opaque code. In *Proc. of the Joint Meeting on Foundations of Software Engineering*. 710–720.

Mark Hills, Paul Klint, and Jurgen Vinju. 2013. An empirical study of PHP feature usage: A static analysis perspective. In *Proc. of the International Symposium on Software Testing and Analysis*. 325–335.

IBM. 2007. IBM Security AppScan. Retrieved from http://www-03.ibm.com/software/products/en/appscan.

IBM Research. 2006. T.J. Watson Libraries for Analysis (WALA). Retrieved from http://wala.sf.net.

Dongseok Jang and Kwang-Moo Choe. 2009. Points-to analysis for JavaScript. In *Proc. of the Symposium on Applied Computing*. 1930–1937.

Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proc. of the Conference on Computer and Communications Security*. 270–283.

Casper S. Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. 2015a. Stateless model checking of event-driven applications. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 57–73.

Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the eval that men do. In *Proc. of the International Symposium on Software Testing and Analysis*. 34–44.

Simon Holm Jensen, Magnus Madsen, and Anders Møller. 2011. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proc. of the Joint Meeting on Foundations of Software Engineering*. 59–69.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *Proc. of the International Symposium on Static Analysis*. 238–255.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2010. Interprocedural analysis with lazy propagation. In *Proc. of the International Symposium on Static Analysis*. 320–339.

Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. 2015b. MemInsight: Platform-independent memory debugging for JavaScript. In *Proc. of the Joint Meeting on Foundations of Software Engineering*. 345–356.

L. Jiang, G. Misherghi, Z. Su, and S. Glondu. 2007. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proc. of the International Conference on Software Engineering*. 96–105.

Richard Joiner, Thomas Reps, Somesh Jha, Mohan Dhawan, and Vinod Ganapathy. 2014. Efficient runtime-enforcement techniques for policy weaving. In *Proc. of the International Symposium on Foundations of Software Engineering*. 224–234.

Seth Just, Alan Cleary, Brandon Shirley, and Christian Hammer. 2011. Information flow analysis for JavaScript. In *Proc. of the International Workshop on Programming Language and Systems Technologies for Internet Clients*. 9–18.

KAIST PLRG. 2012. SAFE: JavaScript analysis framework. Retrieved from http://safe.kaist.ac.kr.

Seonghoon Kang and Sukyoung Ryu. 2012. Formal specification of a JavaScript module system. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 621–638.

Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and Chung-chieh Shan. 2012. An analysis of the Mozilla Jetpack extension framework. In *Proc. of the European Conference on Object-Oriented Programming*. 333–355.

Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A static analysis platform for JavaScript. In *Proc. of the International Symposium on Foundations of Software Engineering*. 121–132.

Vineeth Kashyap and Ben Hardekopf. 2014. Security signature inference for JavaScript-based browser addons. In *Proc. of the International Symposium on Code Generation and Optimization*. 219–229.

Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. 2013. Type refinement for static analysis of JavaScript. In *Proc. of the Symposium on Dynamic Languages*. 17–26.

Matthias Keil and Peter Thiemann. 2013. Type-based dependency analysis for JavaScript. In *Proc. of the Workshop on Programming Languages and Analysis for Security*. 47–58.

Matthias Keil and Peter Thiemann. 2015. TreatJS: Higher-order contracts for JavaScripts. In *Proc. of the European Conference on Object-Oriented Programming*. 28–51.

Gregor Kiczales and Erik Hilsdale. 2001. Aspect-oriented programming. *SIGSOFT Software Engineering Notes* 26, 5 (Sept. 2001).

Haruka Kikuchi, Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. 2008. JavaScript instrumentation in practice. In *Proc. of the Asian Symposium on Programming Languages and Systems*. 326–341.

Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. 2015. Practically tunable static analysis framework for large-scale JavaScript applications. In *Proc. of the International Conference on Automated Software Engineering*. 541–551.

Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proc. of the Conference on Programming Language Design and Implementation*. 278–289.

Hongki Lee, Soocheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *Proc. of the International Workshop on Foundations of Object-Oriented Languages*.

Sungho Lee, Julian Dolby, and Sukyoung Ryu. 2016. HybriDroid: Static analysis framework for android hybrid applications. In *Proc. of the International Conference on Automated Software Engineering*. 250–261.

Benjamin S. Lerner, Matthew J. Carroll, Dan P. Kimmel, Hannah Quay-De La Vallee, and Shriram Krishnamurthi. 2012. Modeling and reasoning about DOM events. In *Proc. of the USENIX Conference on Web Application Development*. 1–12.

Benjamin S. Lerner, Liam Elberty, Jincheng Li, and Shriram Krishnamurthi. 2013. Combining form and function: Static types for jquery programs. In *Proc. of the European Conference on Object-Oriented Programming*. 79–103.

Benjamin S. Lerner, Liam Elberty, Neal Poole, and Shriram Krishnamurthi. 2013b. Verifying web browser extensions' compliance with private-browsing mode. In *Proc. of the European Symposium on Research in Computer Security*. 57–74.

Vladimir I. Levenshtein. 1996. Binary codes capable of correcting deletions, insertions and reversal. *Soviet Physics Doklady* 10, 8 (1996), 707–710.

Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: Automatic symbolic testing of JavaScript web applications. In *Proc. of the International Symposium on Foundations of Software Engineering*. 449–459.

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: A manifesto. *Communications of the ACM* 58, 2 (2015), 44–46.

Francesco Logozzo and Herman Venter. 2010. RATA: Rapid atomic type analysis by abstract interpretation—Application to JavaScript optimization. In *Proc. of the International Conference on Compiler Construction*. 66–83.

Wangzhong Lu, J. Janssen, E. Milios, N. Japkowicz, and Yongzheng Zhang. 2006. Node similarity in the citation graph. *Knowledge and Information Systems* 11, 1 (2006), 105–129.

Magnus Madsen, Benjamin Livshits, and Michael Fanning. 2013. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proc. of the Joint Meeting on Foundations of Software Engineering*. 499–509.

Magnus Madsen and Anders Møller. 2014. Sparse dataflow analysis with pointers and reachability. In *Proc. of the International Symposium on Static Analysis*. 201–218.

Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static analysis of event-driven node.js JavaScript applications. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 505–519.

Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. An operational semantics for JavaScript. In *Proc. of the Asian Symposium on Programming Languages and Systems*. 307–325.

Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2009a. Isolating JavaScript with filters, rewriting, and wrappers. In *Proc. of the European Conference on Research in Computer Security*. 505–522.

Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2009b. Run-time enforcement of secure JavaScript subsets. In *Proc. of the Workshop in Web 2.0 Security & Privacy*.

Sergio Maffeis and Ankur Taly. 2009. Language-based isolation of untrusted JavaScript. In *Proc. of the Computer Security Foundations Symposium*. 77–91.

Josip Maras, Jan Carlson, and Ivica Crnkovi. 2012. Extracting client-side web application code. In *Proc. of the Conference on World Wide Web*. 819–828.

Josip Maras, Maja Štula, and Jan Carlson. 2013. Generating feature usage scenarios in client-side web applications. In *Proc. of the International Conference on Web Engineering*. 186–200.

Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. 2012. Eval begone!: Semi-automated removal of eval from JavaScript programs. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 607–620.

Ali Mesbah, Engin Bozdag, and Arie van Deursen. 2008. Crawling AJAX by inferring user interface state changes. In *Proc. of the International Conference on Web Engineering*. 122–134.

Ali Mesbah and Arie van Deursen. 2009. Invariant-based automatic testing of AJAX user interfaces. In *Proc. of the International Conference on Software Engineering*. 210–220.

Ali Mesbah, Arie van Deursen, and Stefan Lenselink. 2012. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web* 6, 1 (2012), 3:1–3:30.

Bertrand Meyer. 1988. *Object-Oriented Software Construction*. Prentice-Hall.

Leo A. Meyerovich and Benjamin Livshits. 2010. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *Proc. of the Symposium on Security and Privacy*. 481–496.

Microsoft. 2012. TypeScript. Retrieved from http://www.typescriptlang.org.

Amin Milani Fard, Ali Mesbah, and Eric Wohlstadter. 2015. Generating fixtures for JavaScript unit testing. In *Proc. of the International Conference on Automated Software Engineering*. 190–200.

Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. Caja: Safe active content in sanitized JavaScript. Google white paper. Retrieved from. http://google-caja.googlecode.com.

Eric Miraglia. 2007. A JavaScript Module Pattern. Yahoo! User Interface Blog. Retrieved from www.yuiblog.com/blog/2007/06/12/module-pattern/.

Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2013a. Efficient JavaScript mutation testing. In *Proc. of the International Conference on Software Testing, Verification and Validation*. 74–83.

Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2013b. PYTHIA: Generating test cases with oracles for JavaScript applications. In *Proc. of the International Conference on Automated Software Engineering*. 610–615.

Mehdi Mirzaaghaei and Ali Mesbah. 2014. DOM-based test adequacy criteria for web applications. In *Proc. of the International Symposium on Software Testing and Analysis*. 71–81.

Anders Møller, Simon Holm Jensen, Peter Thiemann, Magnus Madsen, Matthias Diehn Ingesman, Peter Jonsson, and Esben Andreasen. 2012. TAJS: Type Analyzer for JavaScript. Retrieved from https://github.com/cs-au-dk/TAJS.

Mozilla. 2013. asm.jsJ. Retrieved from http://asmjs.org.

Mozilla. 2014. Jetpack. Retrieved from http://google-caja.googlecode.com.

Mozilla Developer Network. 2005. About JavaScript. Retrieved from https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript.

James M. Neighbors. 2015. Tutorial: Metacompilers. Retrieved from http://bayfronttechnologies.com/mc_tutorial.html.

Inc. Neo Technology. 2010. Neo4J. Retrieved from http://neo4j.com.

Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proc. of the Conference on Computer and Communications Security*.

James Noble, Sophia Drossopoulou, Mark S. Miller, Toby Murray, and Alex Potanin. 2016. Abstract data types in object-capability systems. In *Proc. of the International Workshop on Aliasing, Capabilities and Ownership*.

Frolin Ocariza, Karthik Pattabiraman, and Ali Mesbah. 2015. Detecting inconsistencies in JavaScript MVC applications. In *Proc. of the International Conference on Software Engineering*. 325–335.

Frolin S. Ocariza, Jr., Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2013. An empirical study of client-side JavaScript bugs. In *International Symposium on Empirical Software Engineering and Measurement*. 55–64.

Frolin S. Ocariza, Jr., Karthik Pattabiraman, and Ali Mesbah. 2012. AutoFLox: An automatic fault localizer for client-side JavaScript. In *Proc. of the International Conference on Software Testing, Verification and Validation*. 31–40.

Frolin S. Ocariza, Jr., Karthik Pattabiraman, and Ali Mesbah. 2014. Vejovis: Suggesting fixes for JavaScript faults. In *Proc. of the International Conference on Software Engineering*. 837–847.

Frolin S. Ocariza, Jr., Karthik Pattabiraman, and Benjamin Zorn. 2011. JavaScript errors in the wild: An empirical study. In *Proc. of the International Symposium on Software Reliability Engineering*. 100–109.

Ejike Ofuonye and James Miller. 2008. Resolving JavaScript vulnerabilities in the browser runtime. In *Proc. of the International Symposium on Software Reliability Engineering*. 57–66.

Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, Daejun Park, Jeehoon Kang, and Kwangkeun Yi. 2014. Global sparse analysis framework. *ACM Transactions on Programming Languages and Systems* 36, 3 (2014), 8:1–8:44.

OWASP. 2013. Types of Cross-Site Scripting. Retrieved from https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proc. of the International Conference on Software Engineering*. 75–84.

Changhee Park, Hongki Lee, and Sukyoung Ryu. 2013. All about the with statement in JavaScript: Removing with statements in JavaScript applications. In *Proc. of the Symposium on Dynamic Languages*.

Changhee Park and Sukyoung Ryu. 2015. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *Proc. of the European Conference on Object-Oriented Programming*. 735–756.

Changhee Park, Sooncheol Won, Joonho Jin, and Sukyoung Ryu. 2015b. Static analysis of JavaScript web applications in the wild via practical DOM modeling. In *Proc. of the International Conference on Automated Software Engineering*. 552–562.

Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2015a. KJS: A complete formal semantics of JavaScript. In *Proc. of the Conference on Programming Language Design and Implementation*. 428–438.

Jihyeok Park. 2014. JavaScript API misuse detection by using typescript. In *Proc. of the International Conference on Modularity (ACM Student Research Competition)*. 11–12.

Joonyoung Park, Inho Lim, and Sukyoung Ryu. 2016. Battles with false positives in static analysis of JavaScript web applications in the wild. In *Proc. of the International Conference on Software Engineering*.

Karthik Pattabiraman and Benjamin Zorn. 2010. DoDOM: Leveraging DOM invariants for web 2.0 application robustness testing. In *Proc. of the International Symposium on Software Reliability Engineering*. 191–200.

Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race detection for web applications. In *Proc. of the Conference on Programming Language Design and Implementation*. 251–262.

Phu H. Phung, David Sands, and Andrey Chudnov. 2009. Lightweight self-protecting JavaScript. In *Proc. of the International Symposium on Information, Computer, and Communications Security*. 47–60.

Jacques A. Pienaar and Robert Hundt. 2013. JSWhiz: Static analysis for JavaScript memory leaks. In *Proc. of the International Symposium on Code Generation and Optimization*. 1–11.

Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. 2011. ADsafety: Type-based verification of JavaScript sandboxing. In *Proc. of the USENIX Conference on Security*. 12–12.

Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. 2012a. Semantics and types for objects with first-class member names. In *Proc. of the International Workshop on Foundations of Object-Oriented Languages*.

Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. 2012b. A tested semantics for getters, setters, and eval in JavaScript. In *Proc. of Symposium on Dynamic Languages*. 1–16.

Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. 2014. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 33–47.

Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *Proc. of the International Conference on Software Engineering*. 314–324.

Michael Pradel and Koushik Sen. 2015. The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript. In *Proc. of the European Conference on Object-Oriented Programming*. 519–541.

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe and efficient gradual typing for typescript. In *Proc. of the Symposium on Principles of Programming Languages*. 167–180.

Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. 2009. NOZZLE: A defense against heap-spraying code injection attacks. In *Proc. of the Conference on USENIX Security Symposium*. 169–186.

Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. 2010. JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. In *Proc. of the USENIX Conference on Web Application Development*. 3–3.

Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from "big code." In *Proc. of the Symposium on Principles of Programming Languages*. 111–124.

Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective race detection for event-driven programs. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 151–166.

Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. 2007. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Transactions on the Web* 1, 3, Article 11 (2007).

John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proc. of the Symposium on Logic in Computer Science*. 55–74.

Dustin Rhodes, Tim Disney, and Cormac Flanagan. 2014. Dynamic detection of object capability violations through model checking. In *Proc. of the ACM Symposium on Dynamic Languages*. 103–112.

Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. 2011a. Automated construction of JavaScript benchmarks. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 677–694.

Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011b. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *Proc. of the European Conference on Object-Oriented Programming*. 52–78.

Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *Proc. of the Conference on Programming Language Design and Implementation*. 1–12.

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete types for typescript. In *Proc. of the European Conference on Object-Oriented Programming*. 76–100.

Grigore Rosu and Traian Florin Serbanuta. 2014. K overview and simple case study. In *Proc. of the International K Workshop (K'11)*, Vol. 304.

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A symbolic execution framework for JavaScript. In *Proc. of the Symposium on Security and Privacy*.

Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic determinacy analysis. In *Proc. of the Conference on Programming Language Design and Implementation*. 165–174.

Matthias Schur, Andreas Roth, and Andreas Zeller. 2013. Mining behavior models from enterprise web applications. In *Proc. of the Joint Meeting on Foundations of Software Engineering*. 422–432.

Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A tool framework for concolic testing, selective record-replay, and dynamic analysis of JavaScript. In *Proc. of the Joint Meeting on Foundations of Software Engineering*. 615–618.

Koushik Sen, George C. Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-path symbolic execution using value summaries. In *Proc. of the Joint Meeting on Foundations of Software Engineering*.

Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation tracking for points-to analysis of JavaScript. In *Proc. of the European Conference on Object-Oriented Programming*.

Vincent St-Amour and Shu-yu Guo. 2015. Optimization coaching for JavaScript. In *Proc. of the European Conference on Object-Oriented Programming*. 271–295.

Louis Stowasser. 2015. CraftyJS: A flexible framework for JavaScript games. Retrieved from http://craftyjs.com.

Dafydd Stuttard and Marcus Pinto. 2011. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Wiley.

Kwangwon Sun and Sukyoung Ryu. 2015. Static Analysis of JavaScript Programs: Challenges and Research Trends (Paper List and Citation Graph). Retrieved from http://plrg.kaist.ac.kr/doku.php?id=research:material.

Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. 2014. Gradual typing embedded securely in JavaScript. In *Proc. of the Symposium on Principles of Programming Languages*. 425–437.

Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. 2011. Automated analysis of security-critical JavaScript APIs. In *Proc. of the Symposium on Security and Privacy*. 363–378.

The jQuery Foundation. 2006. jQuery. Retrieved from http://jquery.com.

Peter Thiemann. 2005. Towards a type system for analyzing javascript programs. In *Proc. of the European Symposium on Programming*. 408–422.

Suresh Thummalapenta, K. Vasanta Lakshmi, Saurabh Sinha, Nishant Sinha, and Satish Chandra. 2013. Guided test generation for web applications. In *Proc. of the International Conference on Software Engineering*. 162–171.

TIOBE Software. 2015. TIOBE Index for November 2015. Retrieved from www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

Omer Tripp, Pietro Ferrara, and Marco Pistoia. 2014. Hybrid security analysis of web JavaScript code via dynamic partial evaluation. In *Proc. of the International Symposium on Software Testing and Analysis*.

Omer Tripp and Omri Weisman. 2011. Hybrid analysis for JavaScript security assessment. In *Proc. of the Joint Meeting on Foundations of Software Engineering*.

Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. 2007. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. of the Network and Distributed System Security Symposium*.

W3C. 1998. Document Object Model Activity Statement. Retrieved from http://www.w3.org/DOM/Activity.

W3C. 2015. UI Events Specification. Retrieved from https://www.w3.org/TR/uievents/.

Shiyi Wei and Barbara G. Ryder. 2013. Practical blended taint analysis for JavaScript. In *Proc. of the International Symposium on Software Testing and Analysis*. 336–346.

Shiyi Wei and Barbara G. Ryder. 2014. State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. In *Proc. of the European Conference on Object-Oriented Programming*. 1–26.

Shiyi Wei and Barbara G. Ryder. 2015. Adaptive context-sensitive analysis for JavaScript. In *Proc. of the European Conference on Object-Oriented Programming*. 712–734.

Shiyi Wei, Franceska Xhakaj, and Barbara G. Ryder. 2016. Empirical study of the dynamic behavior of JavaScript objects. *Software: Practice and Experience* 46, 7 (2016), 867–889.

Yahoo. 2008. ADsafe: Making JavaScript safe for advertising. Retrieved from http://www.adsafe.org.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proc. of the Symposium on Security and Privacy*. 590–604.

Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. 2007. JavaScript instrumentation for browser security. In *Proc. of the Symposium on Principles of Programming Languages*. 237–249.

Chuan Yue and Haining Wang. 2009. Characterizing insecure JavaScript practices on the web. In *Proc. of the International Conference on World Wide Web*. 961–970.

Tian Zhao. 2011. Polymorphic type inference for scripting languages with object extensions. In *Proc. of the Symposium on Dynamic Languages*. 37–50.

Received March 2016; revised May 2017; accepted June 2017