A Survey of Dynamic Analysis and Test Generation for JavaScript

ESBEN ANDREASEN, Aarhus University
LIANG GONG, University of California, Berkeley
ANDERS MØLLER, Aarhus University
MICHAEL PRADEL, TU Darmstadt
MARIJA SELAKOVIC, TU Darmstadt
KOUSHIK SEN, University of California, Berkeley
CRISTIAN-ALEXANDRU STAICU, TU Darmstadt

JavaScript has become one of the most prevalent programming languages. Unfortunately, some of the unique properties that contribute to this popularity also make JavaScript programs prone to errors and difficult for program analyses to reason about. These properties include the highly dynamic nature of the language, a set of unusual language features, a lack of encapsulation mechanisms, and the "no crash" philosophy. This paper surveys dynamic program analysis and test generation techniques for JavaScript targeted at improving the correctness, reliability, performance, security, and privacy of JavaScript-based software.

CCS Concepts: •**General and reference** → **Surveys and overviews;** •**Software and its engineering** → **Software notations and tools;** •**Security and privacy** → *Web application security;*

General Terms: Algorithms, Languages, Performance, Reliability, Security, Verification

Additional Key Words and Phrases: Program analysis, Dynamic languages, Test generation

1. INTRODUCTION

JavaScript has become one of the most prevalent programming languages. Originally, it was designed as a simple scripting language embedded in browsers and intended to implement small scripts that enhance client-side web applications. Over the past two decades, the language has evolved beyond all expectations into one of the most prevalent programming languages. Today, JavaScript is heavily used not only by almost all client-side web applications, but also for mobile applications, desktop applications, and server applications.[1] Many JavaScript programs are complex software projects, including, e.g., highly interactive web sites and online games, integrated development environments, email clients, and word processors.[2] The JavaScript language has been standardized in several versions of the ECMAScript language specification. At the time of this writing, ECMAScript 7 [ECMA 2016] is the latest stable version. We refer to the language as "JavaScript" in the remainder of this article.

---

[1]See, e.g., http://cordova.apache.org/, http://electron.atom.io/, http://nwjs.io/, and http://nodejs.org/.
[2]See, e.g., http://www.y8.com/, http://c9.io/, http://gmail.com/, and https://www.onenote.com/.

Two reasons for the popularity of JavaScript are its dynamic and permissive nature. First, the language is highly dynamic, allowing developers to write code without any type annotations, to extend any objects, including built-in APIs, at any point in time, and to generate and load code at runtime. Second, the language is very permissive with respect to potentially erroneous behavior. Instead of failing an execution when other languages would, JavaScript often follows a "no crash" philosophy and continues to execute. Even when JavaScript code produces a runtime error, e.g., when calling an undefined function, it only aborts the current event handler, allowing the application to proceed. Many developers perceive these two properties—dynamic and permissive—as beneficial for quickly implementing and for easily extending applications.

Unfortunately, the dynamic and permissive nature of JavaScript also causes challenges for producing high quality code. For example, not crashing on likely misbehavior can easily hide an error, allowing errors to remain unnoticed even in production. As another example, dynamic code loading increases the risk that attackers inject and execute malicious code. As a result, JavaScript is known not only for its ease of use but also for being prone to correctness, efficiency, and security problems.

To avoid or at least mitigate such problems, developers need program analyses that detect and help understand errors. However, many traditional program analyses are not effective for JavaScript. First, techniques developed for other widely used languages, such as Java, C, and C++, often cannot be easily adapted because of various JavaScript-specific language features, such as prototype-based inheritance, dynamic typing, and events. Furthermore, the tight interaction of JavaScript code with its environment, e.g., with the DOM in a browser, makes it difficult to adapt existing analyses for other dynamically typed languages, such as Python. Second, static analysis, which is effective for statically typed languages, faces serious limitations in JavaScript because it cannot precisely reason about the many dynamic language features. Instead, there has been a need for novel program analyses that address the challenges created by JavaScript.

These limitations have triggered research on program analysis for JavaScript. In particular, *dynamic analysis* has proven to be an effective way to find and understand problematic code. By reasoning about a running program, dynamic analysis avoids the challenge of statically approximating behavior, a challenge that is particularly difficult for JavaScript. Since dynamic analysis requires inputs that exercise the program, approaches for *test generation* have been developed. These approaches automatically create inputs, such as sequences of UI events or input data given to a function.

The success of dynamic analysis and test generation for JavaScript has led to an abundance of research results, especially in the past decade. While this development is good news for JavaScript developers, the sheer amount of existing work makes it difficult for interested non-experts to understand the state of the art and how to improve upon it. This article addresses the challenge of summarizing the large amount of work on dynamic analysis and test generation for JavaScript. We present a comprehensive survey that enables interested outsiders to get an overview of this thriving research field. In addition to presenting past work, we also outline challenges that are still to be addressed, guiding future work into promising directions.

Figure 1 outlines this article. At first, we summarize challenges imposed by JavaScript (Section 2). Then, we discuss dynamic analyses that help achieve four goals:

— *Correctness and reliability analyses* detect correctness problems and other issues that affect the reliability of a program (Section 3).
— *Security and privacy analyses* detect malicious and vulnerable code (Section 4).
— *Performance analyses* help improve the efficiency of the program (Section 5).
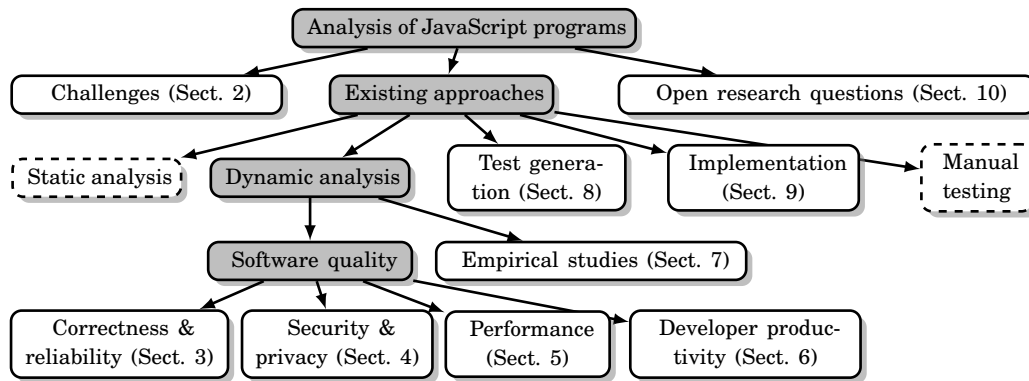
Fig. 1: Overview of topics covered in this article. Each white box corresponds to a section, except for topics in dashed boxes, which are out of the scope of this article.

— *Developer productivity analyses* help programmers during the development process, e.g., during program understanding and program repair (Section 6).

In addition to analyses used by developers, Section 7 discusses empirical studies performed using dynamic analysis. Test generation approaches, in particular approaches to generate input data and sequences of events, are the focus of Section 8. Section 9 compares different ways to implement the approaches surveyed in this article. Finally, Section 10 discusses open research challenges and outlines directions for future work.

Since this article focuses on dynamic analysis and test generation, other related techniques, such as purely static analysis, formalizations and extensions of JavaScript, and manual testing, are out of scope. In practice, the line between static and dynamic analysis is fuzzy because an analysis may interpret the program in a way similar but not equal to its execution on a target platform. We define dynamic analysis as an approach that (i) executes a program on a regular execution platform, i.e., a platform that is also used for executions that do not analyze the program, and that (ii) reasons about individual executions, and not about all possible executions of the program.

There exist several surveys on techniques for JavaScript but, to the best of our knowledge, none covers dynamic analysis and test generation in detail. An inspiring survey by Mesbah [2015] discusses testing of JavaScript-based web applications, with a focus on manual testing, test oracles, and UI-level testing. Our article does not cover manual testing techniques but focuses on automated techniques. Li et al. [2014b] also focus on manual testing but cover neither program analysis nor test generation. Other related work [Garousi et al. 2013; Dogan et al. 2014] performs systematic mapping studies, which focus on a broad classification of approaches. As a contribution over all these existing surveys, we provide a detailed discussion of open research challenges and possible directions for future work.

2. CHALLENGES FOR ANALYZING JAVASCRIPT

This section discusses properties that make JavaScript particularly intricate for program analysis (Section 2.1), explains why these properties naturally lead to dynamic analysis (Section 2.2), and outlines challenges dynamic analysis and test generation for JavaScript face (Section 2.3).

2.1. JavaScript: An Unusual Language

Today's popularity of JavaScript surprises considering how JavaScript came to life: In 1995, a single engineer, Brendan Eich, designed and implemented the first version

in only ten days [Severance 2012]. Initially intended to implement small scripts that enhance web sites, JavaScript now powers applications that reach way beyond what was anticipated. JavaScript is also unusual in its choice of language features:

—*Highly dynamic.* JavaScript is dynamically typed and provides numerous dynamic language features (Section 2.2).
—*Event-driven.* JavaScript uses an asynchronous execution model. Programs consist of event handlers that are triggered by user events and system events.
—*Prototype-based.* JavaScript is object-oriented but not class-based. Instead, each object has a chain of prototype objects to share and reuse code and data.

While some of these features are also present in other widely used languages—e.g., Python is also dynamically typed—those languages that have received most attention by the program analysis community, Java and C, do not provide them.

Another distinguishing property of JavaScript is to be embedded into complex and heterogeneous environments. The most prevalent environment is a browser, where JavaScript interacts with multiple other languages, in particular, HTML and CSS, and with the Document Object Model (DOM). Beyond the browser, server-side JavaScript code typically runs on the Node.js platform, whose API differs from the browser API. Likewise, hybrid mobile applications are typically embedded into an environment that allows for interactions with the mobile device, such as Apache Cordova. Analyzing the interactions of a program with one or more of these environments is challenging. The challenge is compounded by the fact that even environments of the same kind, such as different browsers, often vary in subtle ways across implementations and versions.

Since its creation in 1995, JavaScript has grown considerably by adding new language features. Due to backward compatibility concerns, many of the less fortunate design choices made in the early stages cannot be undone. This increasing complexity of the language also increases the effort required to build analysis tools.

2.2. Why is Dynamic Analysis So Widely Used for JavaScript?

The highly dynamic nature of JavaScript makes it less amenable for static analysis but a well-suited target for dynamic analysis. In the following, we discuss some of the dynamic features of the language and their implications for program analysis.

Types. JavaScript is dynamically typed, i.e., there are no type annotations in the source code. The language uses "duck typing", meaning that whether an object is suitable for an operation depends only on the properties of the object, but not on its nominal type. Furthermore, JavaScript heavily uses implicit type conversions, where a runtime value is converted from one type to another to enable otherwise impossible operations. All these features make it difficult for a purely static analysis to approximate types with reasonable precision, a challenge addressed by various static analyses [Thiemann 2005; Jensen et al. 2009; Guha et al. 2011] not covered in this paper.

Properties and Functions. JavaScript code may dynamically add and remove properties of an object, making it difficult to statically determine which properties are available at a particular program point and what these properties refer to. Since functions are values in JavaScript, the difficulties of statically reasoning about properties also apply to functions. An additional challenge is that a program may dynamically compute a property name before accessing the property. The dynamic nature of properties and functions cause severe difficulties for static analysis [Sridharan et al. 2012; Andreasen and Møller 2014].

Code Loading and Code Generation. Many JavaScript applications load parts of the code at runtime, e.g., via the <script> tag of an HTML page, which causes the browser

to retrieve code from a server, or programmatically via an `XMLHttpRequest`, which loads code asynchronously. Furthermore, JavaScript allows a program to interpret a string as code, and thereby to generate and dynamically load code, e.g., via the built-in `eval` function. These two properties challenge the common assumption of static analysis that the code of the program is readily available.

While these language features challenge static analysis, they reduce to non-issues in dynamic analysis. Because a dynamic analysis observes concrete values and concrete types, it does not suffer from dynamic typing and dynamic usages of properties and functions. Likewise, dynamic analysis is oblivious to dynamic code loading because it analyzes the program's execution once the source code has been loaded. On the flip side, dynamic analysis is inherently limited by the executions it observes and therefore cannot provide soundness guarantees for all possible executions. If soundness is a requirement, e.g., for verifying the absence of security vulnerabilities, static analysis is the more suitable choice. In this article, we focus on dynamic analyses.

2.3. Challenges for Dynamic Analysis and Test Generation

Even though the dynamic features of JavaScript make the language amenable for dynamic analysis, some challenges remain.

"No crash" Philosophy. JavaScript follows a "no crash" philosophy, aiming for running a program without showing any obvious signs of misbehavior, such as a crash, to the user. For example, JavaScript remains silent while executing obviously incorrect operations, such as multiplying an array with a string. The absence of obvious signs of misbehavior makes it difficult for a dynamic analysis to distinguish intended from unintended behavior. Nevertheless, various ways to identify misbehavior have been proposed, as discussed in Sections 3, 4, 5, and 8.

Nondeterminism. The heavy interactions of JavaScript applications with users and the network lead to a high degree of hard-to-control inputs. Since dynamic analysis focuses on individual executions triggered by specific inputs, it may miss behavior triggered by other possible inputs. This input dependence is compounded by several non-traditional sources of nondeterminism, e.g., asynchronously scheduled events whose execution order is nondeterministic. As a result, repeatedly analyzing a program execution driven by a single input may lead to multiple behaviors. We discuss techniques to handle nondeterminism and input dependence, in particular dynamic data race detectors for JavaScript (Section 3) and record-and-replay systems (Section 6).

Event-driven. Due to the event-driven nature of JavaScript programs, only specific sequences of input events may be able to reach a particular region of code. Creating such sequences of events is different from traditional input data generation. Test generation (Section 8) addresses this challenge.

Despite its dynamic nature and sometimes unusual semantics, JavaScript has several properties that make testing and analyzing programs easier than in many other languages. First, the language is essentially single-threaded and therefore does not suffer from concurrency issues caused by shared-memory multi-threading.[3] Second, since source code is the only widely used format to distribute JavaScript programs, there is no need to handle compiled code. Code obfuscation, which is commonly used to distribute proprietary code, does not significantly affect program analyses because obfuscation is semantics-preserving. Finally, testing and analyzing partial code is easier than in statically compiled languages, since an incomplete program may still run.

---

[3]Data races exist nevertheless, as we discuss in Section 3.3.

3. CORRECTNESS AND RELIABILITY

Correctness and reliability are two of the most important challenges faced by JavaScript developers. Correctness here means that a program conforms to its specification. Reliability means the extent to which a software system delivers usable services when those services are demanded. Because of the "no crash" philosophy and the fact that typical JavaScript programs do not come with a correctness specification, there is no clear definition of what constitutes correct execution. Even runtime errors, e.g., dereferencing null, reading from an undeclared variable, or calling a non-function value, may be benign, since uncaught exceptions terminate only the current event handler but not the application. Moreover, JavaScript is memory-safe (i.e., it has automatic garbage collection, arrays cannot be accessed out-of-bound, and there is no pointer arithmetic), so programming errors cannot lead to memory corruption. All this makes the problem of ensuring the correctness and reliability of JavaScript programs different from programs in other popular languages, such as Java and C.

A significant amount of work tries to alleviate correctness issues by detecting them before deploying the software. This section presents such techniques based on dynamic analyses. Specifically, we present approaches that address code smells and bad coding practices (Section 3.1), that target cross-browser issues (Section 3.2), and that detect data races (Section 3.3).

3.1. Code Smells, Bad Coding Practices, and Program Repair

Code smells indicate potential quality issues in the program. To deal with code smells, the software development community has learned over time guidelines and informal rules to help avoid common pitfalls of JavaScript. Those rules specify which language features, programming idioms, APIs, etc. to avoid, or how to use them correctly. Following these rules often improves software quality by reducing bugs, increasing performance, improving maintainability, and preventing security vulnerabilities.

Lint-like tools are static checkers that enforce code practices and report potential code smells. Unfortunately, due to the dynamic nature of JavaScript, approaches for detecting code smells statically are limited in their effectiveness. Although there are some successful static checkers used in real-world application development (e.g., JSHint, JSLint, ESLint, and Closure Linter), they are limited by the need to approximate possible runtime behavior and often cannot precisely determine the presence of a code smell. To address this issue, JSNose [Fard and Mesbah 2013] and DLint [Gong et al. 2015b] dynamically detect code smells missed by existing static lint-like tools.

JSNose [Fard and Mesbah 2013] combines static analysis and dynamic analysis to detect code smells in web applications. The approach mostly focuses on code smells at the level of closures, objects, and functions. For example, JSNose warns about suspiciously long closure chains, unusually large functions, the excessive use of global variables, and not executed and therefore potentially dead code. In contrast, DLint [Gong et al. 2015b] detects violations of coding practices at the level of basic JavaScript operations, such as local variable reads and writes, object property reads and writes, and function calls. The approach monitors these operations and detects instances of bad coding practices by checking a set of predicates at runtime. Gong et al. also report an empirical study that compares the effectiveness of DLint's checkers and their corresponding static checkers in JSHint. The result of the study suggests that dynamic checking complements static checkers: Some violations of coding practices can be detected only by a dynamic checkers, e.g., because the violation depends on a runtime type, whereas other violations are bound only by a static checker, e.g., because it depends the syntactic structure of the code.

TypeDevil [Pradel et al. 2015] addresses a particular kind of bad practice: type inconsistencies. They arise when a single variable or property holds values of multiple, incompatible types. To find type inconsistencies, the analysis records runtime type information for each variable, property, and function, and then merges structurally equivalent types. By analyzing program-specific type information, TypeDevil detects problems missed by checkers of generic coding rules, such as DLint and JSNose.

Some analyses not only detect problems but also help fixing them. One such analysis, Evalorizer [Meawad et al. 2012], replaces unnecessary uses of the `eval` function with safer alternatives. Using this function is discouraged because its subtle semantics is easily misunderstood, because it has a negative performance impact, and because `eval` may enable attackers to execute untrusted code. Evalorizer dynamically intercepts arguments passed to `eval` and transforms the `eval` call to a statement or expression without `eval`, based on a set of rules. The approach assumes that a call site of `eval` always receives the same or very similar JavaScript code as its argument.

Vejovis [Ocariza Jr. et al. 2014] generates fixes for bugs caused by calling the DOM query API methods, e.g., `getElementById`, with wrong parameters. The approach identifies possible bugs by dynamically checking whether a DOM query returns abnormal elements, e.g., `undefined`, or whether there is an out-of-range access on the returned list of elements. Based on the observed symptoms and the current DOM structure, Vejovis suggests fixes, such as passing another string to a DOM method or adding a `null`/`undefined` check before using a value retrieved from the DOM.

3.2. Cross-browser Testing

Because supported language features and their implementation may differ across browsers, cross-browser compatibility issues challenge client-side web applications. Such issues are caused by ambiguities in the language specification, or by disagreements among browser vendors. Incompatibilities often lead to unexpected behavior.

CrossT [Mesbah and Prasad 2011] detects cross-browser issues by first crawling the web application in different browsers to summarize the behavior into a finite state model, and then finding inconsistencies between the generated models. WebDiff [Choudhary et al. 2010b; 2010a] structurally matches components in the DOM trees generated in different browsers and then computes the visual difference of screenshots of the matching components. In addition to comparing captured state models and comparing visual appearances, CrossCheck [Choudhary et al. 2012] incorporates machine learning techniques to find visual discrepancies between DOM elements rendered in different browsers. Based on their previous work, Choudhary et al. further proposed X-PERT [Choudhary et al. 2013; 2014b], which detects cross-browser incompatibilities related to the structure and content of the DOM and to the web site's behavior. The tool compares the text content of matching components and the relative layouts of elements by extracting a relative alignment graph of DOM elements. As X-PERT addresses limitations of previous work by the same authors, it seems to be the most comprehensive approach for detecting cross-browser issues. FMAP [Choudhary et al. 2014a; Choudhary 2014] aims at detecting missing features between the desktop version and the mobile version of a web application by collecting and comparing their network traces. The approach assumes that the web application uses the same back-end functionality while having specific customizations in the front-end.

3.3. Dynamic Race Detection

Even though JavaScript programs execute in a single thread and each event handler runs without preemption, there may be races, as known from concurrent programs. The reason is that the ordering of responses to asynchronous requests, timer events, script and frame loading, and HTML parsing is not fully controlled by the developer,

which may introduce nondeterminism. For example, a common mistake is to assume that all HTML parsing has completed before any user event handlers are executed, or that network communication is synchronous. As a consequence, users with a slow network connection or a different browser than the developer may experience UI glitches and data corruption. Because JavaScript provides no built-in mechanisms for concurrency control programmers resort to ad hoc synchronization, such as timer events that repeatedly postpone processing until some flag is set by another event handler. The problem with races in JavaScript was first described by Steen [2009] and Ide et al. [2009]. An early static approach to detect races [Zheng et al. 2011] faces not only the challenges with static analysis for JavaScript, but its practicability is limited by not taking the possible event orderings into account.

WebRacer by Petrov et al. [2012] pioneered dynamic race detection for JavaScript. In particular, they characterize the happens-before relation that models the causal relationship of events. The results from WebRacer show that races are prevalent, even in widely used web sites. However, not all races correspond to errors. First, ad hoc synchronization inevitably causes races although its purpose is to prevent errors. Second, for event handlers that commute, different event orderings may lead to the same state. Third, even when different event orderings lead to different states, those differences may be benign, e.g., if they do not affect the UI or network data. For this reason, much of the later work has focused on improving precision. The follow-up tool EventRacer [Raychev et al. 2013] introduced the notion of race coverage to filter many harmless races caused, e.g., by ad hoc synchronization.

Instead of looking for races per se, Wave [Hong et al. 2014] heuristically explores different event schedules and checks whether the final DOM states differ. While exploring schedules, Wave respects the happens-before relation and does not change the sequence of user events. The potential advantage is that it directly targets the consequences of the races. However, as noted by Jensen et al. [2015a], event handlers influence each other, so aggressively reordering events to provoke races often leads to infeasible event sequences, in which case Wave falsely reports errors.

The R^4 algorithm by Jensen et al. [2015a] aims to systematically explore different event schedules to discover the consequences of races. It takes a sequence of events from an initial execution as input. Unlike Wave, R^4 uses conflict-reversal bounding and approximate replay for reducing divergence from the initial execution. Moreover, by adapting dynamic partial order reduction, R^4 avoids exploring many equivalent schedules. The approach significantly reduces the number of false positives compared to Wave, but overall still suffers from a non-negligible amount of false positives.

Mutlu et al. [2015] argue that most races are harmless and focus on those that affect persistent storage, either on the client or on the server. Instead of actually executing alternative event schedules like Wave and R^4, their approach is to perform a lightweight static analysis on the possible schedules for the given execution to see whether persistent storage may be affected.

Although the problem with races is by now well described and multiple detection techniques have been developed, the accuracy of the available tools are not yet sufficient for production use. It remains an open question how to automatically and effectively detect races that are harmful and explain their consequences to developers.

4. SECURITY AND PRIVACY

JavaScript powers many applications with strong security and privacy requirements, such as online banking, shared editing of private documents, and e-commerce. As a result, various program analyses address security and privacy issues. These efforts are a response to the limitations of the current security mechanisms: they either relax

these mechanisms or enhance them by defending against newly identified threats. As surveyed in previous work [Ryck et al. 2010], the goals of the research community with respect to the security of JavaScript applications are manifold: separate scripts either from each other or from the underlying system, mediate interactions between scripts and communication of scripts with other entities, and fine grained control of behavior. Considering the large amount of JavaScript-related work on security and privacy, this section focuses only on the last category. Specifically, we discuss *dynamic information flow analyses*, an area that has received considerable attention from the program analysis community, in particular for JavaScript.

Information flow analysis is a heavyweight technique that reasons about each executed instruction. The analysis assigns security labels to values and propagates these labels through the program. A label expresses the security level of the value, e.g., how secret a value is. The analysis checks whether the flows of values complies with a security policy that specifies where values with a particular label are allowed to flow to. Depending on the security policy, information flow analysis can detect different kinds of vulnerabilities and malicious behavior. Unless explicitly specified, the analyses discussed here can be used with various policies. For a general survey of information flow techniques for other languages the reader is directed to [Sabelfeld and Myers 2003].

4.1. Taint Analysis

At first, we discuss work on taint analysis, a lightweight form of information flow analysis that considers only data flows. All analyses discussed in this subsection address cross-site scripting (XSS) vulnerabilities. These are vulnerabilities where user-controlled inputs may modify the DOM, e.g., by adding a `script` tag with additional JavaScript code. Lekies et al. [2013] conduct a study that detects thousands of DOM-based XSS vulnerabilities across the Alexa top 5,000 web sites. To validate each detected vulnerability, an exploit is generated. The prevalence of sophisticated DOM-based vulnerabilities observed in popular web sites demands more powerful mechanisms for protecting the users. The traditional way to detect these problems is to use string-based XSS filters. Stock et al. [2014] describe and address the inability of the existing XSS filters to defend against DOM-based XSS attacks. They propose a dynamic taint engine that tracks the origin of strings and prevents the interpretation of user-provided input into tokens other than literals. This restrictive policy leads to a low false positive rate, and it detects five times more true positives than traditional, string-based XSS filters.

Hybrid taint analysis approaches combine static and dynamic analysis. Partial evaluation using the recorded dynamic data is proposed by Tripp et al. [2014] as a way to increase the precision of a static analysis. Their analysis rewrites DOM accesses into abstractions that the static taint analysis can reason about. The approach significantly reduces false positives compared to a purely static analysis. Another hybrid approach [Wei and Ryder 2013] guides static analysis with dynamically recorded traces. In particular, this technique makes dynamically generated code visible to the static analysis. The approach detects vulnerabilities that purely static taint analysis misses, while being more efficient. However, it is unknown how the hybrid approach compares to purely dynamic approaches. The two presented hybrid solutions differ in the information they collect at runtime, but share the idea of providing dynamically gathered information to a static analysis.

Once a taint analysis detects that a flow violates the given security policy, a major challenge is to decide whether the violation is a security problem or a false positive. Saxena et al. [2010b] address this challenge by combining a taint analysis with fuzzing to generate attack strings against potential validation vulnerabilities. The analysis is performed on a trace that summarizes the JavaScript execution in a language with

simpler semantics. The candidate exploits are further validated using a browser-based oracle.

4.2. Information Flow Analysis

Dynamic information flow analysis for JavaScript has received significant attention because several language features make this problem particularly hard: how to reason about the large number of non-JavaScript APIs (DOM and other web APIs), the prevalence of `eval`, prototype inheritance, and the special scoping rules of `with` blocks. Existing work aims at tackling subsets of these problem, while keeping the overhead and the implementation effort at an acceptable level. One additional, language-independent challenge for dynamic information flow analyses is how to handle flows caused by *not* executing a branch. In this work we refer to this type of flows as *hidden flows*. They represent an additional information flow channel created by the runtime labels specific to purely dynamic analyses [Sabelfeld and Myers 2003].

Austin and Flanagan [2010] introduced the first purely dynamic information flow analysis, called a monitor, for a core of JavaScript. It provides two monitoring strategies, *No Sensitive Upgrade* (NSU) and *Permissive Upgrade* (PU), both ensuring noninterference. These strategies prevent implicit leaks of information by stopping the execution of the program whenever the monitor reaches an unsafe state. To enhance the permissiveness of these techniques, developers may insert upgrade operations that aid the monitor from stopping the execution unnecessarily. Hedin and Sabelfeld [2012] have generalized these ideas to a larger subset of JavaScript that includes objects, higher-order functions, exceptions, and dynamic code evaluation. Hedin et al. [2014] implemented this approach in a custom interpreter that supports the full language, with models for built-in APIs and the browser environment. Chudnov and Naumann [2015] proposed a rewriting-based, inlined monitor that implements the NSU policy for full ECMAScript 5 with web support. The evaluation shows that the monitor is able to run one order of magnitude faster than the one by Hedin et al. [2014] and that it can enforce information flow control on synthetic mashups. However, since the inliner is implemented in Haskell, it cannot protect against dynamically generated code. The above approaches rely on upgrade operations that improve the permissiveness of the information flow analysis. An approach for automatically inserting upgrade statements [Birgisson et al. 2012] combines testing and program rewriting for transforming hidden flows into explicit flows. The main drawback is that the permissiveness of the monitor depends on the completeness test suite.

A completely different approach [Austin and Flanagan 2012] takes inspiration from secure multi-execution [Devriese and Piessens 2010], a technique for enforcing security policies through multiple, concurrent runs of the program. The key idea are *faceted values*, i.e., runtime objects that contain multiple alternative values, one for each security level. Each action in a program is associated with a principal that corresponds to a security level. Whenever an output operation is reached, the facet corresponding to the principal's security level is passed to the sink. The technique handles hidden flows by construction, and for simple policies and large numbers of principals it scales better than the original work on secure multi-execution.

Bichhawat et al. [2014] describe a hybrid information flow analysis that handles hidden flows. To deal with unstructured control flow, the approach performs an on-demand static analysis. Chugh et al. [2009] propose another hybrid method in which a static information flow analysis is first performed on the server. This stage outputs a set of constraints to be checked on the client-side for the dynamically loaded code, reducing the client-side overhead. This reduction comes at a small loss of precision due to lack of context-sensitivity. Even though these two presented techniques are

Publications	Type	Explicit flows	Implicit flows	Hidden flows
Saxena et al. 2010b	dynamic	✓		
Lekies et al. 2013	dynamic	✓		
Stock et al. 2014	dynamic	✓		
Tripp et al. 2014	hybrid	✓		
Wei and Ryder 2013	hybrid	✓		
Austin and Flanagan 2010	dynamic	✓	✓	disabled
Austin and Flanagan 2012	dynamic	✓	✓	disabled
Hedin and Sabelfeld 2012	dynamic	✓	✓	disabled
Hedin et al. 2014	dynamic	✓	✓	disabled
Dhawan and Ganapathy 2009	dynamic	✓	✓	
Chudnov and Naumann 2015	dynamic	✓	✓	
Birgisson et al. 2012	dynamic	✓	✓	✓
Bichhawat et al. 2014	hybrid	✓	✓	✓
Chugh et al. 2009	hybrid	✓	✓	✓

Table I: Comparison of considered information flows.

both hybrid, they differ in their objectives: one aims at handling hidden flows and the other one at reducing the client-side overhead.

Dhawan and Ganapathy [2009] propose a fine grained, purely dynamic information flow analysis for a specialized JavaScript environment: the Firefox browser extensions. The approach concentrates on confidentiality-violating policies, monitoring flows from the DOM and cross-domain accesses to the network and the file system. The analysis correctly detects known malicious behavior in four extensions.

To summarize the information flow analyses surveyed in this section, Table I classifies all analyses. The table summarizes whether an approach is purely dynamic or hybrid, i.e., static and dynamic. Furthermore, the table shows which kinds of information flow each analysis considers. Explicit flows refer to information flows caused by data flows. Implicit flows consist of runtime observed control flows, while hidden flows are information flows caused by not executing a write instruction in an alternative branch. Some analysis disable hidden flows by stopping the program before such a flow may appear. As can be observed, the analyses vary from basic dynamic taint analysis to complex hybrid analysis that trigger static analysis on demand. Which analysis to pick for a particular purpose depends, e.g., on the attack model considered by an approach, on the level of runtime overhead that is considered acceptable, and on the strength of the desired security guarantees.

5. PERFORMANCE

Because JavaScript was initially implemented through interpretation, it was long been perceived as a "slow" language. The increasing complexity of applications created a need to execute JavaScript code more efficiently. Due to various efforts, the performance has since been tremendously improved, which in turn has enabled entirely new kinds of applications. This section presents these efforts, focusing on approaches that involve dynamic analysis. In particular, we discuss improvements of runtime systems (Section 5.1), browser-independent analyses to identify known performance issues and code smells (Section 5.2), and advances on benchmarks for performance measurement (Section 5.3).

5.1. Enhancements of JIT Compilers

Just-in-time (JIT) compilation has a long history of improving the execution time of a program by compiling it to efficient machine code at runtime [Aycock 2003], and most modern JavaScript engines use JIT compilation. For example, according to St-Amour et al. [St-Amour and Guo 2015], the SpiderMonkey JavaScript engine first interprets code without any compilation or optimization. Upon reaching a specific number of executions of a function, the baseline JIT compiler translates the function to native code. Once a function becomes hot, the optimizing compiler further optimizes it based on gathered runtime observations. The current V8 engine (version 5.6) skips the interpretation phase entirely and instead compiles JavaScript code directly to native code. Hot functions in the compiled code are further optimized at runtime. Optimization techniques used in JavaScript JIT compilers include inlining, dead code elimination, and type specialization.

Despite producing efficient machine code, the optimizations applied by JIT engines are limited due to optimistic assumptions that the compiler makes before optimizing the code. In cases when these assumptions become invalid, the compiler throws away optimized code, and resumes execution with generic code. This process is known as *deoptimization*, which is an inherently expensive operation. For example, a JIT compiler may specialize code for particular types or values observed in the past, and will have to deoptimize the code if other types or values occur. To reduce or even avoid deoptimizations, JIT compilers use sophisticated dynamic analyses to understand which runtime behavior is likely to occur in the future. Key to the success of such analyses is to be efficient, because the runtime cost of analyzing the program must outweigh the benefit obtained by optimizing the program.

In the rest of this section we briefly discusses approaches that have been developed specifically for JavaScript and that involve a non-trivial dynamic analysis.

5.1.1. Type Specialization. Due to the dynamically typed nature of JavaScript, JIT compilers do not have access to static type information. This lack of information makes the generation of efficient, type-specialized machine code difficult. TraceMonkey by Gal et al. [2009] was one of the first JIT compilers for JavaScript. Based on the observation that programs spend most of the time in hot loops and that most loops are type-stable, the compiler specializes the code for frequently executed loops at runtime. At the core of TraceMonkey is a dynamic analysis that gathers sequences of statements, called traces, along with their type information. These traces may cross function boundaries. The analysis represents frequently executed traces in a tree structure that encodes the type conditions under which the code can be specialized. The experimental results show that loops have few combinations of variable types and loop specialization yields to a significant performance improvement of JavaScript programs over the baseline interpreter. Their work suggests further improvements by specializing recursive function calls, regular expressions, and expression evaluation to avoid interpreting these language constructs.

Despite the initial success of trace-based JIT compilation, the approach has since mostly been abandoned, e.g., at the favor of hybrid (static and dynamic) type inference [Hackett and Guo 2012]. In this hybrid approach, a static analysis computes for each expression or heap value, a possibly incomplete set of types it may have at runtime. At runtime, the JIT engine checks for unexpected types and other special cases, such as arrays containing undefined values and integer overflows.

Type specialization in JIT compilers is speculative, and when unexpected types are encountered, the compiler deoptimizes type-specialized code. To reduce the number of deoptimizations, Kedlaya et al. [2015] propose ahead-of-time profiling on the server side. Their profiler tracks when a function becomes hot, which types and shapes a func-

tion uses, and when it gets deoptimized. Based on information about type-unstable functions, the client-side engine prevents optimizing code that will likely be deoptimized later. A limitation of their approach is to focus on a specific set of deoptimizations.

5.1.2. Function Specialization. In contrast to the above approaches, which exploits dynamically observed types, de Assis Costa et al. [2013] propose to specialize functions based on dynamically observed values. Their approach is based on the empirical observation that 60% of all JavaScript functions are called only once or always with the same set of parameters. Based on this observation, they propose a JIT optimization that replaces the arguments passed to a function by previously observed runtime values. A limitation of their approach is to consider primitive values only. Future work may extend their ideas to objects.

5.2. Refactoring Approaches to Address Performance Issues in JavaScript Code

The efficiency of a JavaScript program is affected not only by the runtime engine, but naturally also by the programming patterns, algorithms, and language constructs used in the program. In the following, we discuss approaches that identify performance bottlenecks in JavaScript code based on their symptoms, such as code fragments that prohibit JIT optimizations or cause memory leaks. Furthermore, these approaches also provide suggestions for refactorings to improve the performance of JavaScript code.

5.2.1. JIT-unfriendly Code. JITProf by Gong et al. [2015a] is an engine-independent profiling approach that identifies code locations that prohibit profitable JIT optimizations. Such code locations are called JIT-unfriendly. The general idea is to simulate the execution of a JIT compiler by associating meta-information with JavaScript objects and code locations that are updated whenever particular runtime event occurs. This meta-information is later used to identify JIT-unfriendly code locations. JITProf is an extensible framework that allows developers to specify their own patterns of JIT-unfriendly code. Being engine-independent, the approach relies on an accurate simulation of the JIT engine's behavior, which may change in a way that makes supposedly JIT-unfriendly code harmless in future engines.

A similar work by Xiao et al. [2015] is JSweeter, an approach that also finds JIT-unfriendly code locations but focuses on performance code smells related to type mutations. They instrument the V8 engine to collect type update operations and deoptimizations. The collected information is used to infer the reasons and number of deoptimizations, eventually reporting type-unstable code locations. JITProf and JSweeter both provide refactoring hints on how to optimize the code, but they consider different JIT-unfriendly patterns.

Another approach to help developers find JIT-unfriendly code it to focus on optimizations that could almost be applied but that the compiler cannot apply due to lack of information or potential unsoundness. Optimization coaching by St-Amour and Guo [2015] searches for such missed optimizations. The approach considers two optimizations: accesses of property and elements, as well as assignments. All missed optimizations that affect the same operation or originate from the same type of failure are merged and ranked based on the expected performance impact. Similar to JITProf and JSweeter, the optimization coach recommends program changes that trigger additional optimizations.

A limitation of all existing approaches to find JIT-unfriendly code is to rely on built-in knowledge of JIT-optimizations and JIT-unfriendly code patterns. Since JavaScript engines evolve quickly, future engines are likely to suffer from different patterns. It remains an open question how to automatically identify JIT-unfriendly code patterns.

Another promising direction, which is motivated by the non-homogeneity of JavaScript engines, is to detect engine-independent performance problems.

5.2.2. Memory-Related Issues. MemInsight by Jensen et al. [2015b] is a browser-independent memory debugging tool for web applications that computes object lifetimes. The approach generates a trace of memory operations during an execution, capturing the uses of each object, variable declarations, and return calls. The trace is then used to recover object lifetime information by simulating the application's heap. Several client analyses use this information to find memory leaks, drags, churns, and opportunities for stack allocation and object inlining. The evaluation shows that the tool can expose unknown memory issues in several real-world applications. To detect memory problems, MemInsight uses several heuristics, which may give false warnings. Another limitation of the approach is that it is effective only for long running applications.

5.3. Performance Benchmarks

Vendors of JavaScript engines demonstrate the performance of engines by running benchmarks. The most commonly used JavaScript benchmark suites are SunSpider[4], Octane[5], and Kraken[6]. Unfortunately, many of these benchmarks turn out to be not representative of real-world code, as shown by Ratanaworabhan et al. [2010] and discussed further in Section 7. As an implication, focusing on benchmark behavior may result in overfitting and missing optimization opportunities that are present in real applications. Some of these benchmark suites have been updated with several real-world programs and tests for measuring new aspects of JavaScript performance, such as compiler latency and garbage collection.[7]

Motivated by the lack of representativeness of existing benchmarks, Richards et al. [2011a] developed JSBench to automate the creation of realistic and representative JavaScript benchmarks from existing web applications. JSBench instruments the original web code to generate a trace of JavaScript operations. The trace is used to generate a replayable JavaScript program while replacing nondeterministic call sites and achieving high fidelity. Finally, the program is recombined with HTML from the original web application. Richards et al. show that JSBench-generated benchmarks match the behavior of real web applications by using several metrics, such as memory usage, GC time, and event loop behavior, collected on several instrumented browsers.

All existing benchmarks focus on the language defined in the ECMAScript 5 specification. Section 10.6 discusses challenges related to evolving benchmarks toward newer language versions.

6. DEVELOPER PRODUCTIVITY

Developers of JavaScript applications face several challenges during the development process that are not strictly related to correctness, performance, and security. This section presents and compares dynamic program analyses aimed at making developers more productive, beyond the approaches discussed in Section 3 to 5.

At first, this section clusters analyses based on their primary purpose, and discusses and compares related approaches with each other. We identify three main groups of analyses aimed at making developers more productive:

---

[4]https://webkit.org/perf/sunspider/sunspider.html

[5]https://developers.google.com/octane/

[6]http://krakenbenchmark.mozilla.org/

[7]http://browserbench.org/JetStream/

—*Record and replay* enable developers to capture and re-execute (parts of) an execution, e.g., for debugging (Section 6.1).

—*Visualizations* of recorded executions can help understand the interactions between different parts of a program (Section 6.2).

—*Slicing* identifies a subset of the program's statements responsible for creating a particular value (Section 6.3).

At the core of many of these approaches are dynamic analysis techniques that keep track of dependencies between program elements and runtime operations. Section 6.5 compares these techniques with each other, showing that different purposes require different techniques for dynamic dependency tracking.

6.1. Record and Replay

Record and replay techniques use dynamic analysis to capture a program execution for future replay. Possible applications include debugging, i.e., finding the root cause of a failure, and automated testing, i.e., automatically executing a sequence of user events for regression testing. The following discussion covers approaches that target client-side web applications, which is the main focus of existing work.

Mugshot [Mickens et al. 2010] and Dolos [Burg et al. 2013] address the challenge that the execution of a JavaScript program may depend on nondeterministic behavior, such as the order of asynchronously scheduled events, the order and content of incoming network responses, UI events triggered by the user, and inherently nondeterministic behavior, such as random number generation. Both approaches aim at capturing all possible sources of nondeterminism into a trace, which enables deterministic replay. For example, Mugshot and Dolos intercept calls to `setTimeout` to keep track of when the callback function passed to `setTimeout` is invoked, and to replay the callback at the same point in (logical) time. The two approaches differ in how they capture sources of nondeterminism. Mugshot injects JavaScript code into the application so that this code executes before any of the application code. The injected code overwrites built-in functions with wrappers that log calls to them. In contrast, Dolos modifies the WebKit JavaScript engine to capture calls related to nondeterministic behavior. Compared to Mugshot, the Dolos approach has the benefit of providing more condensed traces, e.g., because a single click event may trigger multiple event handlers, each of which Mugshot will record. In contrast, a clear benefit of Mugshot is to be applicable across browsers.

Jalangi [Sen et al. 2013] stands out by supporting "selective" record and replay, i.e., the ability to record and replay the execution of specific parts of the program only. This feature allows users to exclude, e.g., third-party libraries from being considered. Jalangi tracks all memory reads and their values, except those that can be re-computed based on already recorded values by re-executing the corresponding code. The approach is implemented in an engine-independent way via source-level instrumentation. Even though Jalangi's initial focus has been record and replay, it is now most widely used as a dynamic analysis framework (Section 9).

A more lightweight form of record and replay is to focus on sequences of UI events. Selenium[8] is a widely used implementation of such an approach. It is a browser automation tool that allows testers to capture an interaction with a web application into a UI-level test case, and to replay it afterwards. A major challenge is to capture all UI events, because missed events will lead to incomplete test cases that may not be replayed accurately. WaRR [Andrica and Candea 2011] addresses this challenge by modifying the Chrome browser so that it captures and replays a large set of UI events.

----

[8]http://www.seleniumhq.org/

According to Andrica and Candea [2011], WaRR can replay interactions between users and complex web applications that Selenium fails to support, such as drag-and-drop and editing documents in a shared, web-based editor. Neither Selenium nor WaRR capture sources of nondeterminism beyond events triggered by the user. As a result, these approaches may not be able to accurately replay a recorded event sequence because the application diverges from the behavior it exhibited during recording.

An understudied challenge related to record and replay approaches is how to adapt recorded information when the underlying program evolves. For example, it is desirable to use Selenium-recorded interactions for regression testing. However, these interactions may become invalid when the application changes in subtle ways, e.g., by renaming a DOM element locator that the recorded interaction relies on to find an element. Existing work on evolving UI-level tests [Grechanik et al. 2009; Zhang et al. 2013; Leotta et al. 2014] partially addresses this challenge, but does not consider the specific problems of JavaScript-based applications.

6.2. Visualization for Program Understanding

Understanding the interactions between different parts of a client-side JavaScript application is nontrivial. One reason are interactions between artifacts written in different languages and stored in different files, namely JavaScript code, HTML code, and CSS stylesheets. Another reason is the event-driven nature of JavaScript, which makes it difficult to understand what causes an event. To help developers understand an execution, several techniques for visualizing executions have been proposed. A commonality of all these approaches is to capture runtime events with a dynamic analysis and to present the events to the user as a timeline. In addition to the temporal order of events, several approaches also visualize the causal relationships of events and link events to those parts of the JavaScript code that causes the particular behavior.

Some approaches allow a user to indicate a particular DOM element of interest, which is useful when investigating how a particular behavior is implemented. Fire-Crystal [Oney and Myers 2009] tracks all DOM changes related to the selected DOM element and presents them along with the responsible JavaScript, HTML, and CSS code. Unravel [Hibschman and Zhang 2015] instead tracks and visualizes the function call tree that causes a DOM change. Another approach [Burg et al. 2015] also checks which DOM mutation operations cause a change in the visual appearance of the DOM element of interest and lets the user focus on those operations.

Clematis [Alimadadi et al. 2014] visualizes an execution at several levels of detail and lets users zoom into particular parts of the behavior. In contrast to the approaches discussed above, Clematis captures calls that register asynchronous callback functions to track the causal dependencies between event handlers. Two existing approaches go beyond the client-side part of a web application and instead capture and visualize the interactions between client-side and server-side. FireDetective [Matthijssen et al. 2010] focuses on Java-based server-side implementations and shows what JavaScript code triggers what Java code via asynchronous requests. Sahand [Alimadadi et al. 2016] addresses applications where both the client-side and the server-side are implemented in JavaScript. The approach summarizes an execution into a directed graph that represents function executions and their causal relationships.

Profilers shipped as part of the developer tools of popular browsers, e.g., the Chrome or Firefox developer tools, provide performance-related visualizations. For example, Chrome provides a timeline tool that records which functions execute when, and then plots this information over a time axis.

Going beyond interactions between a single and a single server, future work on visualizing JavaScript-based applications could consider interactions that involve multiple clients and multiple server components. Another promising direction is to not

only summarize what has happened in an execution, but to also highlight potential misbehavior, e.g., based on known anti-patterns.

6.3. Program Slicing

Program slicing extracts a subset of a program's statements relevant for computing a particular value, which is useful, e.g., for understanding the behavior of the program. Similar to information flow analysis (Section 4), slicing requires to track dependencies, but it focuses on dependencies between statements instead of dependencies between values. In principle, traditional dynamic slicing approaches [Agrawal and Horgan 1990] can be adapted to JavaScript. One challenge is how to handle interactions of JavaScript code with the DOM. Maras et al. [Maras et al. 2011; 2012] address this problem to build a dynamic analysis that, given a particular DOM element of interest and a user demonstrating how to use it, extracts all code related to implementing this DOM element's functionality. The analysis first records the execution path and then interprets the executed statements while creating a dependency graph that encodes structural relationships, data flow dependencies, and control flow dependencies between HTML elements, JavaScript code locations, and parts of CSS specifications. Based on this graph, slicing the code that influences the DOM node of interest is equivalent to computing which nodes are reachable from a statement of interest.

AutoFLox [Ocariza Jr. et al. 2012] gathers an execution trace and uses it to compute a subset of the program's statements responsible for passing an `undefined` or `null` value into a DOM method that triggers an exception. The approach is based on dynamic backward slicing that tracks the incorrect value back to its origin. Compared to Maras et al.'s work, AutoFLox does not explicitly represent dependencies involving HTML and CSS elements but instead considers dependencies between JavaScript statements only.

6.4. Other Work on Improving Programmer Productivity

To help developers who write code that interacts with the DOM, Dompletion [Bajaj et al. 2014] provides code completion for strings passed into the DOM API. To find suitable suggestions, the tool dynamically analyzes the DOM and JavaScript code that interacts with it. The challenge of executing different paths through the program is addressed by analyzing individual functions in isolation while forcing particular paths to be executed.

When a program evolves, developers would like to understand which statements are impacted by a particular change. Tochal [Alimadadi et al. 2015] addresses this challenge through a static and dynamic change impact analysis. The main idea is to extract the impact relationships between functions, DOM elements, and XHR objects. To this end, Tochal statically computes a call graph and dynamically extracts how functions influence or are influenced by DOM elements and XHR objects. Tochal may miss impacted code and may report code as impacted even though it is independent of a change. Future work may strive for providing stronger guarantees, such as certainly finding all impacted code or reporting only code that is definitely impacted. Another promising direction for future work is to visualize the impact of a change, e.g., by highlighting the possibly affected parts of the DOM.

Crowdie by Madsen et al. [2016] extracts a control flow path, called a crash path, that is likely to explain the root cause of a crash. The main contribution is to distribute the runtime overhead required for constructing crash paths across multiple users and executions, while iteratively refining the set of dynamically analyzed functions based on feedback from previous executions. Preliminary experiments show that crash paths are helpful in debugging, but the approach has not yet been tested in production.

Publications	Tool names	Control flow	Data flow	Async	DOM
Oney and Myers 2009	FireCrystal	full path		✓	✓
Mickens et al. 2010; Andrica and Candea 2011; Burg et al. 2013	Mugshot, WaRR, Timelapse			✓	✓
Matthijssen et al. 2010; Alimadadi et al. 2014; Alimadadi et al. 2016	FireDetective, Clematis, Sahand	calls		✓	✓
Maras et al. 2011	FireCrow	full path	value provenance		✓
Maras et al. 2012		full path	yes		✓
Ocariza Jr. et al. 2012	AutoFLox	full path	assignments	✓	✓
Sen et al. 2013	Jalangi	full path	memory reads		✓
Bajaj et al. 2014	Dompletion	forced			✓
Burg et al. 2015	Scry	stack traces			✓
Hibschman and Zhang 2015	Unravel				✓
Alimadadi et al. 2015	Tochal		DOM accesses	✓	✓
Madsen et al. 2016	Crowdie	on demand		✓	

Table II: Comparison of dependencies tracked during dynamic analysis.

6.5. Comparison of Approaches to Track Dependencies

Most approaches presented in this section (Section 6) share the idea of tracking dependencies between particular program elements and events during the program execution. In the following, we compare which kinds of dependencies are tracked by the different approaches. This comparison is useful for developers of future analyses, who need to understand what dependencies exist and decide which ones to track.

Table II summarizes the results of our comparison. We consider the four most common kinds of dependencies that analyses track:

— *Control flow.* There are different ways to gather information about the flow of control during the execution. Some approaches keep track of all executed statements, allowing them to exactly reconstruct the execution path ("full path"). Other approaches obtain stack traces at particular points during the execution ("stack traces"), keep track of caller-callee relationships ("calls"), force the execution along a particular path ("forced"), or continuously refine what control flow information to gather ("on demand"). Several approaches do not track control flow at all ("no").

— *Data flow.* None of the approaches perform a full data flow analysis, but several do track some form of data flow: "value provenance" means to keep track of the code location where a value is defined; "assignments" keeps track of data propagated through direct assignments, but ignores data flows via complex expressions; "DOM accesses" means that the approach tracks reads and writes of DOM properties; "strings" means that the analysis focuses on data flows of string values; "memory reads" means that the approach tracks sufficient information to reproduce all memory reads. The fact that no approach performs a full data flow analysis is likely to be motivated by the high overhead that tracking all data flows would impose.

— *Asynchronous functions.* About half of the approaches track the causal relationships between code that registers an asynchronous callback function and the asynchronously executed callback. To track these dependencies, the approaches typically monitor methods that register callbacks, such as `setTimeout` and `XMLHttpRequest.onreadystatechange` and executions of the callback functions.

	Yue and Wang 2009	Jang et al. 2010	Ratanaworabhan et al. 2010	Richards et al. 2010	Richards et al. 2011b	Ocariza Jr. et al. 2011	Nikiforakis et al. 2012	Son and Shmatikov 2013	Behfarshad and Mesbah 2013	Nederlof et al. 2014	Pradel and Sen 2015	Wei et al. 2016	Selakovic and Pradel 2016
Experimental setup:													
Subjects:													
Client-side	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Server-side													✓
Benchmarks			✓	✓								✓	
Method:													
Dynamic analysis	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Static analysis	✓				✓								
Threats to validity					✓	✓			✓	✓	✓	✓	✓
Topic of study:													
Dynamic behavior:													
Objects & properties			✓								✓		
Functions & calls			✓	✓									
Inheritance & prototypes			✓								✓		
Types											✓	✓	
Code inclusion & generation	✓			✓	✓		✓						
DOM									✓	✓			
Other	✓		✓	✓									✓
Code quality issues						✓							✓
Security	✓	✓					✓	✓					
Evolution							✓						✓

Table III: Overview of empirical studies of JavaScript programs.

—*DOM usage.* Almost all approaches presented in this section track accesses to some subset of the DOM API, e.g., to monitor which UI events are triggered. The exception is Jalangi, which, however, can analyze arbitrary program behavior, including DOM usage, while replaying an execution.

In summary, the comparison shows a wide range of options for tracking dependencies. Since dynamic dependency tracking involves both implementation effort and runtime overhead, each approach picks the dependencies that are most appropriate for the respective usage scenario. For example, all record and replay techniques track control flow dependencies because knowing control flow decisions is a prerequisite for replaying an execution.

7. EMPIRICAL STUDIES

Researchers seek to address real-world problems while making realistic assumptions. To help the program analysis community achieve these goals, several empirical studies of real-world JavaScript programs and related artifacts have been performed. This section summarizes the findings of these studies (Section 7.1), discusses their implications and how they have influenced the community (Section 7.2), compares their methodologies (Section 7.3), and outlines areas of interest that are currently understudied and may be addressed in future work (Section 7.2.2). Table III gives an overview of the discussed studies, their methodologies, and the questions they address. We focus our discussion on empirical studies that use dynamic analysis and ignore studies based on purely static analysis [Karim et al. 2012; Gallaba et al. 2015] or purely manual analysis [Ocariza Jr. et al. 2013].

7.1. Results of Studies

7.1.1. Dynamic Language Features. Several studies analyze how particular language features of JavaScript, in particular those that are more dynamic than in other languages, are used in practice. This question is relevant to validate assumptions made by static analyses and for performance optimizations (Section 5). The following summarizes the most important findings:

— *Objects and properties*. Richards et al. [2010] show that, even though most object properties are initialized while constructing the object, it is also common to add properties to or delete properties from objects afterwards. Wei et al. [2016] also investigate the dynamism of objects and show that the average number of own properties per object increases from 28 just after object construction to up to 200 at the end of an object's lifetime. They also report that developers sometimes attempt to delete properties that are not present in the object. One possible explanation of this high degree of dynamism is that objects are often used as map data structures and that objects include arrays, where adding and removing properties is common.
— *Functions and calls*. Functions in JavaScript are variadic, i.e., they can be called with a variable number of arguments. Richards et al. [2010] show that almost 10% of functions use this language feature. Furthermore, their study shows that 19% of all call sites are polymorphic, i.e., different calls done at the same code location invoke different functions. Ratanaworabhan et al. [2010] show that 50–70% of all functions that are statically declared are never executed, a property commonly exploited by JIT compilers (Section 5).
— *Inheritance and prototypes*. The studies show that the use of JavaScript's prototype-based inheritance differs significantly from inheritance in statically typed, class-based languages, such as Java. For example, Richards et al. report that prototype chains often change during an object's lifetime, especially because libraries extend or modify built-in types. They also observe that the length of most prototype chains is very short, with a median of one and a maximum of ten. Wei et al. further observe that most prototype objects (64%) are used as the immediate prototype of only one user object. They conclude that JavaScript programs often do not use prototypes for code reuse, which differs from the use of classes in class-based object-oriented languages, such as Java and C++.
— *Types*. Pradel and Sen [2015] study the use of type coercions, i.e., the implicit conversion of a value of one type into a value of another type. They find that coercions are highly prevalent—over 80% of all function executions perform at least one coercion—but at the same time, mostly harmless and likely intentional. Wei et al. investigate how often the types of properties change at runtime; they report that 99% of all properties of user objects never change their type, yet some properties undergo up to ten type changes during their lifetime.
— *Representativeness of benchmarks*. Several studies question whether commonly used benchmarks, such as SunSpider and Octane, are representative for real-world web sites. Ratanaworabhan et al. [2010] show that many benchmarks differ significantly from web sites, e.g., with respect to the overall code size, the mix of executed instructions, the prevalence of cold code, the typical duration of function executions, and the prevalence of event handlers. Richards et al. confirm several of these findings, and in addition show that the degree of function polymorphism and the distribution of kinds of allocated objects differ. Pradel and Sen report that most benchmarks have significantly fewer type coercions than real web sites.

7.1.2. Code Inclusion and Dynamic Code Generation. Some studies investigate to what extent JavaScript programs include code from third parties and dynamically generate

code at runtime. These questions are particularly relevant for estimating the security impact of third-party code on JavaScript applications.

Code Inclusion. Yue and Wang report that 66% of all analyzed web sites include code from external domains into the top-level document. The study by Nikiforakis et al. [2012] reports that 88% of all sites include at least one remote library. One explanation for the difference between these numbers may be that the study by Nikiforakis et al. has been performed later and that, as shown by them, around 50% of all sites include at least one additional library per year. Nikiforakis et al. also study the prevalence of potentially unsafe ways of code inclusion and find that a small percentage of all web sites load third-party code from a specific IP address instead of using a domain name.

Dynamic Code Generation. Dynamic code loading, e.g., using the `eval` function, is frequently referred to as a feature that distinguishes JavaScript and its analysis from other popular languages. Yue and Wang [2009] show that close to half of all analyzed web sites use `eval`, a result later confirmed by Richards et al. [2010]. Both studies find that many usages of dynamic code loading could be refactored into safer code, such as, simply omitting the `eval` call around an expression that JavaScript would evaluate anyway. Richards et al. propose a taxonomy of scenarios where `eval` is used and study these scenarios in detail.

7.1.3. Runtime Failures and Performance Issues. To steer work on program analyses toward problems that developers face in practice, two studies investigate runtime failures and performance issues in real-world software. Ocariza Jr. et al. [2011] find that many popular web sites trigger runtime failures, in particular "permission denied" errors, which occur when the same-origin policy is violated, and "undefined symbol" errors, which occur when referring to a non-existing function, property, or variable. They also show that many failures occur nondeterministically, depending on how fast user events, such as clicking on a button, are triggered. Selakovic and Pradel [2016] study performance issues reported in open-source projects and how the developers address them. Their study shows that many issues are due to a relatively small number of recurring root causes, such as inefficient API usages, that most issues can be fixed with relatively simple changes, and that the performance gain obtained with an optimization may vary significantly depending on the JavaScript engine. Some of the challenges reported by these studies are already addressed by existing analyses, e.g., by data race detectors (Section 3.3) and performance analysis tools (Section 5.2), but many still wait to be addressed by a suitable analysis.

7.1.4. Security. The (in)security of web sites is subject of several empirical studies. Jang et al. [2010] study the prevalence of supposedly unwanted information flows that leak private user data, e.g., through cookie stealing, location hijacking, and history sniffing. The study shows such leaks to exist in the wild: e.g., 485 of the 50,000 studied web sites gather parts of the user's browsing history by exploiting the fact that browsers render links to already visited pages differently than links to not yet visited pages. Section 4 discusses dynamic analyses targeted at detecting unwanted information flows. Another study [Son and Shmatikov 2013] focuses on security problems related to the `postMessage` API, which enables web sites to elude the same-origin policy. The study shows that 22% of all analyzed sites use this API and that 71% of them do not perform any origin check of messages received via `postMessage`. A relatively small but non-negligible percentage of all sites could be exploited because of such missing origin checks.

7.1.5. Domain Object Model (DOM). The importance of the DOM for client-side JavaScript applications has motivated two studies on how the DOM is used in practice.

Behfarshad and Mesbah [2013] study DOM states, called "hidden states" that are not reachable through a link but only by triggering a JavaScript event. They report that 62% of all DOM states are hidden states and that the most common path into such a state are clicks on `div` elements. This finding may help web crawlers to heuristically uncover more of those states. Nederlof et al. [2014] find that significant parts of the DOM are modified by the browser or by the site's JavaScript code after the page has finished loading, even when the user does not trigger any event. This finding suggests that analyses of the DOM done should not assume that DOM remains stable after some point in time.

7.2. Implications of Studies

Most empirical studies target the program analysis community and aim at conclusions relevant for analysis developers. The implications suggested by these empirical studies fall into two broad categories: challenging assumptions that are commonly made and pinpointing problems that are currently understudied but deserve more attention.

7.2.1. Challenging Commonly Made Assumptions. Studies of how JavaScript is used in practice have contradicted several assumptions made in earlier research efforts. For example, earlier work has often assumed that, even though JavaScript provides various dynamic language features, most code does not use these features extensively. However, studies show that dynamic behavior is highly prevalent:

— Dynamic code loading, e.g., through `eval`, is prevalent [Yue and Wang 2009; Richards et al. 2010].
— Dynamically adding and removing object properties after an object has been constructed is common [Richards et al. 2010; Wei et al. 2016].
— Prototype hierarchies often change at runtime [Richards et al. 2010; Wei et al. 2016].
— Type coercions are highly prevalent [Pradel and Sen 2015].

These findings are relevant for developers of static and dynamic analyses, as well as developers of JIT compilers, because they help align newly developed techniques with the properties of real-world code.

Several studies challenge the assumption that commonly used benchmarks are representative for real-world code, as we discuss above. This finding has motivated the development of novel techniques to obtain more realistic benchmarks [Richards et al. 2011a] (Section 5).

Another assumption challenged by studies is that a particular language feature or API is dangerous. For example, several studies show that many uses of the supposedly harmful `eval` are benign and could be easily removed to avoid the confusions and performance penalties that `eval` can cause [Yue and Wang 2009; Richards et al. 2011b]. This result has triggered work on semi-automatically removing `eval` calls [Meawad et al. 2012]. Another study shows that most type coercions are harmless and should not be treated as errors when type checking JavaScript [Pradel and Sen 2015].

Finally, Behfarshad and Mesbah [2013] challenge the assumption that following links is sufficient to crawl most parts of the web. Several approaches on UI-level testing address this finding (Section 8).

7.2.2. Pinpointing Currently Understudied Problems. Empirical studies often serve as a source of inspiration for problems to address in future program analyses. Several studies pinpoint such opportunities, only some of which have already been addressed. Challenges that developers commonly face include how to reuse third-party code in a secure way [Yue and Wang 2009; Nikiforakis et al. 2012], how to check whether using the `postMessage` API opens the door for content injections [Son and Shmatikov 2013], how to deal with dynamic types and type coercions [Richards et al. 2010; Pradel and

Sen 2015], how to test for nondeterministically occurring runtime failures [Ocariza Jr. et al. 2011], and how to avoid common performance bottlenecks, such as inefficient API usages [Selakovic and Pradel 2016]. The latter three challenges are, at least partially, addressed by analyses discussed in Sections 3.1, 3.3, and 5.2. A study by Jang et al. [2010] poses the question how end users can control or at least observe which private data leaks to web site providers, a problem addressed by approaches discussed in Section 4.

7.3. Comparison of Methodologies

Understanding the methodologies used by existing empirical studies is important for two reasons. First, it helps understand the validity of the findings reported by a study. Second, it helps researchers who conduct future studies to understand the space of possible methodologies and not yet covered methodologies. The upper part of Table III summarizes the methodologies uses by existing empirical studies. Most studies focus on client-side web applications, which are typically selected based on their popularity. The number of analyzed web sites ranges from 11 [Ratanaworabhan et al. 2010] to 50,000 [Jang et al. 2010], with a median of 300. We selected empirical studies that use dynamic analysis; in addition, two studies use a lightweight form of static analysis of dynamically extracted code [Yue and Wang 2009; Richards et al. 2011b].

Seven of the 13 studies explicitly discuss threats to the validity of the conclusions drawn in the study. The most commonly discussed threats are (i) that the subject programs may not be representative for all JavaScript code (discussed six times) and (ii) that the studies are performed with a limited set of browsers or JavaScript engines (discussed four times). These threats naturally occur in any study. To mitigate their effects, most studies use either popular or randomly sampled subjects, and they focus on widely used execution platforms.

8. AUTOMATED TEST GENERATION

This section surveys techniques to automatically create inputs that drive an execution and to generate test assertions that capture the expected behavior of the program. We do not cover general web crawling techniques that have a black-box view on the application code, like Crawljax [Mesbah et al. 2012], even though they exercise JavaScript indirectly, but focus on techniques that directly involve the JavaScript code. The broader topic is surveyed by Mesbah [2015]. We also omit test generation for races from this section since it is treated in Section 3.3.

We classify approaches by separating the input space of an application into two separate, but interacting, spaces: the *event space* and the *value space* [Saxena et al. 2010a]. The event space concerns the order in which event handlers are executed, e.g., the order of clicking on buttons. The value space concerns the more classic notion of input: the choice of values, such as strings, numbers, objects, as well as values written, e.g., into text fields and cookies. Another challenging aspect of test generation is how to produce meaningful assertions. Table IV provides an overview of the test generation techniques discussed in this section.

8.1. Exploration of the Event Space

Kudzu [Saxena et al. 2010a] explores the event space by randomly executing event handler, the conclusion is that this often complements the value space exploration significantly. Artemis [Artzi et al. 2011] use a heuristic search to explore the event space. The heuristic is based on the observed read and write operations by each event handler in an attempt to exclude sequences of non-interacting event handler executions. EventBreak [Pradel et al. 2014] measures the performance cost of an event handler in terms of the number of conditionals it evaluates. This measure is used to search

Publication	Name	Based on	Value space	Event space	Assertions
Saxena et al. 2010a	Kudzu		✓	✓	
Artzi et al. 2011	Artemis		✓	✓	
Heidegger and Thiemann 2012	JSConTest		✓		✓
Mirshokraie and Mesbah 2012	JSart	Crawljax			✓
Mirshokraie et al. 2013	Pythia	Crawljax			✓
Sen et al. 2013	Jalangi		✓		
Fard et al. 2014	Testilizer	Crawljax			✓
Pradel et al. 2014	EventBreak			✓	
Li et al. 2014a	SymJS	Artemis	✓		
Fard et al. 2015	ConFix		✓		
Mirshokraie et al. 2015	JSeft	Crawljax			✓
Sen et al. 2015	MultiSE	Jalangi			
Tanida et al. 2015		Artemis	✓		
Dhok et al. 2016		Jalangi	✓		

Table IV: Overview of automated test generation tools for JavaScript. The three columns *Value space*, *Event space*, and *Assertions* indicate if the tool address that problem area.

for pairs of event handlers that exhibit unbounded, increasing performance costs. The existence of such a slow-down pair indicates potential unresponsiveness of the tested application.

8.2. Exploration of the Value Space

8.2.1. Concolic Execution. Several techniques use concolic execution [Godefroid et al. 2005] for systematically exploring the value space. In order to find client-side code injection vulnerabilities, Kudzu [Saxena et al. 2010a] employs a string constraint solver to systematically explore the value space using concolic execution. One of the client analyses of Jalangi [Sen et al. 2013] is a concolic execution engine, which explores the value space using a linear integer constraint solver for conditions on numbers and strings. The types of input values are heuristically chosen based on their immediate use. Dhok et al. [2016] observe that the dynamically typed nature of JavaScript programs may cause this approach to generate redundant inputs for type tests at branches. They propose type-awareness, and improve the performance of Jalangi's by distinguishing regular path constraints and constraints on types. MultiSE [Sen et al. 2015] also improves the performance of the concolic execution engine of Jalangi, but the technique is not focused on JavaScript.

SymJS [Li et al. 2014a] improves the value space exploration of Artemis with a concolic execution engine. Tanida et al. [2015] improves the automation of SymJS by creating symbolic inputs based on manual type annotations, hereby side-stepping some of the problems of missing type-awareness.

ConFix [Fard et al. 2015] generates HTML fixtures during concolic execution. Their observation is that some program paths depend on the structure of DOM structures, leading to complex constraints that classic constraint solvers are unable to solve. To address this limitation, ConFix collects constraints as XPath expressions and uses a *structural* solver that emits appropriate HTML fixtures that drive the execution.

8.2.2. Other Approaches. JSConTest [Heidegger and Thiemann 2012] provides a contract language for annotating JavaScript functions. Type contracts for function parameters guide the random exploration of the value space: the annotated type indicates the kind of random values to generate, as also done by Tanida et al. [2015]. Similar to Artemis, the choice of random values is informed by literal values in the function body.

We note that test generation is not the primary purpose of the contract language, but an application of annotations written for other purposes.

8.3. Assertion Generation

The problem of generating test assertions, also called the "oracle problem" [Miller and Howden 1981], is orthogonal to exploring the value and event space. An example of this orthogonality is seen in JSConTest [Heidegger and Thiemann 2012], which not only uses contracts for test generation but also for generating runtime assertions, where a failing assertion indicates a violated contract.

JSart [Mirshokraie and Mesbah 2012] generates likely assertions for catching regression errors, using Crawljax for event space exploration and Daikon [Ernst et al. 2007] for assertion generation. Pythia [Mirshokraie et al. 2013] also generates assertions based on a Crawljax-based exploration, but the generated assertions assert that the behaviors of fault-mutated functions do *not* occur. The event space exploration is feedback-directed and greedily explores paths likely to lead to higher function coverage. JSeft [Mirshokraie et al. 2015] supersedes this work with several improvements and two ways of generating assertions. The first kind of generated assertion is for the DOM structures that are relevant for a sequence of events, where the relevance criteria makes the tests less brittle. The second kind of generated assertion is post-conditions at the individual function level. A post-condition asserts that calls to a function with equivalent entry-point states produce equivalent exit-point states. In both cases, the relevant DOM structures and the entry- and exit-point states are chosen based on runtime observations. Testilizer [Fard et al. 2014] is another technique for generating assertions based on Crawljax. The exploration of both event and value space is driven by small variations of the execution paths taken by existing test cases. The small variations enables the generation of assertions that are either copies or minor variations of assertion in the existing tests.

9. IMPLEMENTATIONS OF DYNAMIC ANALYSES AND TEST GENERATORS

9.1. Dynamic Analyses

A dynamic analysis observes an execution of a program and analyzes the observations made during the execution. To observe an execution, dynamic analyses usually implement instrumentation techniques that can be classified into three broad categories.

9.1.1. Runtime Instrumentation. Runtime instrumentation modifies a JavaScript engine to collect runtime information. Most JavaScript engines compile JavaScript code to an intermediate representation and then interpret the instructions in the intermediate representation one-by-one or translate them to machine code. To collect runtime information, a runtime instrumentor modifies the interpreter of the intermediate representation or the code that translates the intermediate representation to machine code. An instrumentor needs to understand the internal representation of the JavaScript program state to collect state information.

9.1.2. Source Code Instrumentation. Source code instrumentation modifies the source code of a program to insert additional code that performs the dynamic analysis. One approach is to insert callbacks that get invoked when the modified program executes. A dynamic analysis implements these callbacks to collect runtime information, such as the name and value of a variable being read, the operation being performed on two operands and the value of those operands. The callbacks are inserted in such a way that they do not change the behavior of the program. The most notable dynamic analysis infrastructure that uses source code instrumentation and callbacks is Jalangi [Sen et al. 2013], which has been the basis for several analyses discussed in this article.

One of the key challenges in source code instrumentation is that the injected code could use a library that is itself is instrumented by the instrumentor. This could lead to unbounded recursive function calls when the instrumented program is executed. To avoid such unbounded recursive calls, Jalangi requires that programmers of analyses port a library used by the analyses to a private namespace. For example, if an analysis needs to use jquery, the programmer could load jquery in a way such that the variables jquery and $ are created not in the global but in a private namespace.

Yu et al. [2007] propose a lightweight source code instrumentation framework to regulate the behavior of untrusted code. Their approach specifies instrumentation using rewrite rules. Since they focus on enforcing security policies, instrumented code only monitors a subset of JavaScript runtime behaviors. Kikuchi et al. [2008] further extend and implement the approach based on a web proxy that intercepts and instruments JavaScript code before it reaches the browser.

Source-code instrumentation frameworks are often built on top of existing AST creation and transformation tools, such as Esprima, Acorn, Estraverse, Escodegen, and Babel[9]. For example, Jalangi [Sen et al. 2013] uses Acorn to parse source code into ASTs, and uses Escodegen to generate instrumented code from transformed ASTs. The newest version of Babel also has a plugin architecture that allows for performing AST transformations, including source code instrumentation.

A key advantage of source code instrumentation over runtime instrumentation is that it requires no modification of a JavaScript engine. Modifying a JavaScript engine is problematic 1) because engines have complex implementations, i.e., any modification requires a lot of engineering effort and 2) because engines evolve rapidly, making it difficult to maintain an analysis. In contrast, source code instrumentation has the limitation that it cannot analyze code that is not instrumented, such as native function calls. Moreover, it is difficult to entirely avoid changing the behavior of the program being instrumented. For example, a stack trace associated with an exception could get polluted with instrumentation information, or a program could convert the body of a function to a string, which would differ from the uninstrumented function. Runtime instrumentors, on the other hand, have two key advantages over source code instrumentors: 1) They can collect full runtime information from an execution irrespective of whether the program calls native functions or not. 2) Runtime instrumentation generally runs faster than code instrumented at source-code level.

9.1.3. Meta-circular Interpreter. A meta-circular interpreter functions in a completely different way from the above two instrumentation techniques—it implements an interpreter of JavaScript in JavaScript. The meta-circular interpreter utilizes the object representation of the underlying interpreter to represent the state of the JavaScript program. It also delegates the native calls made in the JavaScript program to the underlying interpreter. A dynamic analysis is implemented by modifying the behavior of the meta-circular interpreter. The approach is portable as it requires no modification of a JavaScript engine. Moreover, it gives total control and visibility over the execution of a JavaScript program; therefore, it does not suffer from the limitations of source-level instrumentation. Meta-circular interpreters have two disadvantages: 1) They require a faithful implementation of the JavaScript semantics, which is difficult in practice. 2) They cannot perform just-in-time compilation, which tends to slow down execution of the JavaScript programs. A notable meta-circular interpreter that has been used for dynamic analysis is Photon [Lavoie et al. 2014].

---

[9]http://esprima.org/, https://github.com/ternjs/acorn, https://github.com/estools/estraverse, https://github.com/estools/escodegen, https://babeljs.io/

A dynamic analysis framework usually provides two mechanisms to maintain meta-information about runtime values: *shadow values* [Sen et al. 2013; Christophe et al. 2016] and *shadow memory* [Patil et al. 2010]. The shadow value mechanism enables an analysis to associate an analysis-specific meta-value with any value used in a program execution, e.g., taint information or a symbolic representation of the actual value. In contrast, shadow memory associates a meta-value with every memory location used by an execution. Since both runtime instrumentation and meta-circular interpretation controls memory allocation and object layout, they can be modified easily to implement both shadow memory and shadow value mechanisms. Implementation of shadow memory requires the runtime to allocate a shadow object for every actual JavaScript object and a shadow activation frame for every actual activation frame. A shadow value mechanism can be implemented by associating a meta-pointer to each value; the pointer points to the shadow value of the value. For source code instrumentation, shadow memory can be implemented by allocating a shadow object with every object and by creating a shadow activation frame on every function invocation. A shadow value can be associated with every JavaScript value of type `object` or `function`, either by adding a hidden property referencing the shadow object or by mapping the actual object to the shadow object using a weak hash map [Jensen et al. 2015b]. However, this simple approach cannot be used to associate shadow values with primitive types, such as numbers, booleans, and strings. Jalangi [Sen et al. 2013] supports shadow values for primitive values using a record-replay mechanism.

Beyond the three main implementation approaches described above, some dynamic analyses use more lightweight techniques. One such approach exploits the dynamic nature of the language and its APIs by overwriting particular built-in APIs before any other code is executed. The overwriting function then records when the overwritten function is called and forwards the call to the original implementation. Another approach is to register for runtime events via the debugging interface of a JavaScript engine. These lightweight implementation techniques provide only limited access to runtime events, but they are relatively simple to implement.

9.2. Test Generators

Three frameworks are re-used to implement test generators: Artemis [Artzi et al. 2011], Crawljax [Mesbah et al. 2012], and Jalangi [Sen et al. 2013] (Table IV). Artemis and Crawljax both generate sequences of events, such as clicks. To this end, both frameworks check which events are available and let a configurable strategy decide which event to trigger next. Artemis also is an instance of the "runtime instrumentation" category of Section 9.1. The first version of Artemis was based on the Rhino JavaScript engine,[10] with a JavaScript-based model of the DOM. Because of limitations of that DOM model, Artemis has since been ported to a modified version of the WebKit JavaScript engine,[11] with a native DOM implementation.

The three frameworks use different implementation techniques. An Artemis analysis can control and monitor all behaviors of the browser, but this comes at the cost of portability. Conversely, Jalangi provides a portable solution that enables an analysis to control and monitor most behaviors of the JavaScript runtime. Finally, Crawljax enables an analysis to specify how the crawler should interact with DOM elements, and on what high level browser events the analysis should be notified. All three frameworks are open source, with public issue trackers. Crawljax and Jalangi have a plugin-based architecture, providing an easy way for others to build upon these frameworks.
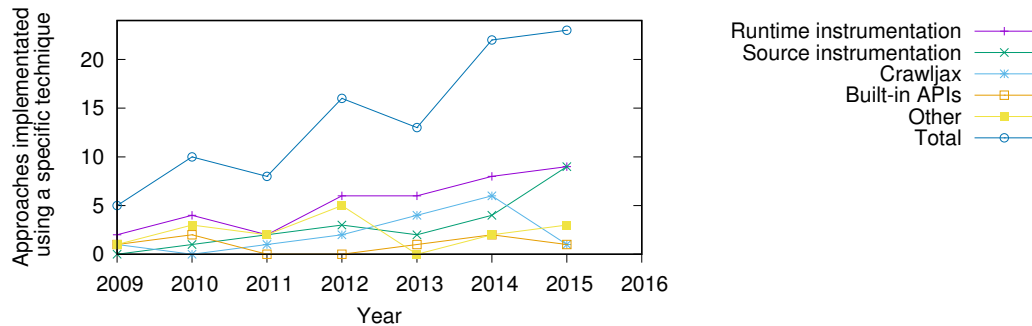
---

[10]https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
[11]https://webkit.org/

Fig. 2: Prevalence of different implementation techniques over time.

9.3. Prevalence and Discussion

There is a wide range of options for implementing dynamic analyses and test generators. Figure 2 summarizes how prevalent the options discussed in this section are among the approaches discussed in this article. Since the beginning, runtime instrumentation has been the most frequently used technique. The majority of implementations build either upon the WebKit JavaScript engine or the Firefox SpiderMonkey engine, with 30 and 18 of all 76 runtime instrumentation-based approaches, respectively. The second-most popular option is source code instrumentation, where 16 of 42 approaches are implemented on top of Jalangi. In contrast, meta-circular interpreters and other implementation techniques play a relatively small role. Among the techniques to implement test generators, Crawljax is the by far most prevalent, accounting for a total of 30 approaches. Figure 2 also shows that the total number of analyses and test generators for JavaScript has been steadily increasing over the past years. Even though the numbers for 2016 cannot yet be finalized at the time of this writing, we do not expect this trend to stop in the near future.

The choice of an implementation strategy is based on the kinds of information and control required by an approach. For a dynamic analysis, the kind of runtime events to be observed determines the available options. For example, for a lightweight analysis interested in observing calls to a fixed set of APIs, simply overwriting these APIs is an easy to implement strategy. For analyses that need to observe all operations of the analyzed program, source code instrumentation and runtime instrumentation are more suitable. The latter is particular useful for analyzing not only JavaScript code but also how the JavaScript code interacts with code implemented by the execution platform, e.g., with built-in APIs implemented through native code. For example, all existing data race detectors (Section 3.3) use runtime instrumentation, because they need to analyze or even control events emitted by the browser, such as network events.

10. OPEN PROBLEMS AND RESEARCH DIRECTIONS

In addition to open questions outlined in the previous sections, this section presents several under-explored research directions that could have a significant impact on the dynamic analysis and test generation communities.

10.1. Combinations of Dynamic Analysis and Test Generation

The two strands of research covered in this article, dynamic analysis and test generation, are obviously related to each other. In particular, automatically generated tests can serve as a driver to execute applications during a dynamic analysis. Despite this potentially fruitful combination, most dynamic analyses are applied either with man-

ually written tests or by manually exploring the analyzed program. An interesting challenge is how to effectively exploit the power of test generation as a driver for a particular dynamic analysis. For example, it would be desirable to create tests that trigger behavior of interest to the dynamic analysis, such as potentially incorrect, inefficient, or insecure code locations. In contrast to this open challenge, the inverse direction of combining dynamic analysis and test generation has already been explored: For example, Artemis [Artzi et al. 2011] and EventBreak [Pradel et al. 2014] (Section 8) feed information obtained via a dynamic analysis back to a test generator, e.g., to trigger not yet covered code.

10.2. Barriers to Adopting Analysis Tools

Despite the increasing interest of the research community in dynamic analysis, its prevalence in industry is not yet on par with static analysis tools [Bessey et al. 2010; Ayewah and Pugh 2010; Sadowski et al. 2015]. A notable exception is the JITCoach tool[12] developed at Mozilla, which has been inspired by JITProf [Gong et al. 2015a]. One challenge is how to reduce the overhead of a dynamic analysis while making it easy to deploy. Currently, low-overhead analyses typically require to modify the runtime environment, e.g., the browser, in an analysis-specific way (Section 9), which makes it hard to deploy the analysis to a wide range of users. Another challenge is to improve the usability of dynamic analysis tools by reducing the amount of false positives, or ideally, remove them altogether. As discussed in Section 3, the notion of correctness is not always clear in real-world JavaScript programs, making it difficult for an analysis to identify undesired behavior. Future research should focus on coming up with stronger oracles that decide when misbehavior occurs. Analyzing the user-perceived effect of potential misbehavior, such as done, e.g., by R^4 [Jensen et al. 2015a] (Section 3.3) is a promising first step in this direction. Finally, visualization could also help improve the usability of results from a dynamic analysis. For example, to help developers understand whether some potentially harmful program behavior, such as a data race, is indeed harmful, visualization of the situation that leads to the behavior may help understand the impact.

10.3. Diverse and Evolving Execution Environments

JavaScript code is deployed in a multitude of execution environments that differ in their support for language features, as well as their implementation and availability of APIs. Furthermore, these execution environments constantly evolve. Novel language features and APIs, and even entirely new execution environments, appear frequently. Based on these observations, we identify three research directions that deserve additional attention.

Execution Platforms Beyond the Browser. Given the strong focus of existing work on client-side web applications executed in a browser, there is a need for analyses that address the specific challenges of other platforms:

—*Mobile platforms*. Some mobile platforms, e.g., Tizen and Firefox OS, support native applications written in JavaScript; others, e.g., Apache Cordova[13], provide a virtualization layer on top of system calls. Important challenges here are the protection of user data from the variety of sensors that the mobile devices expose and the demand for energy efficiency. To our knowledge there is little work addressing the particularities of mobile platforms, with work by Jin et al. [2014] and Lee et al. [2016] being

---

[12]https://github.com/stamourv/jit-coach

[13]See https://www.tizen.org/, https://www.mozilla.org/firefox/os/, and http://cordova.apache.org/.

noteworthy exceptions. Considering the increasing trend towards use of the HTML standards in the mobile platforms there is likely to be demand for additional work.

—*Server-side web applications*. Implementing not only the client-side but also the server-side of a web application in JavaScript is becoming increasingly popular in the last years.[14] Challenges in this environment include how to scale JavaScript code to respond to billions of clients a day and how to defend it against the security issues that characterize the web.

—*Stand-alone applications*. Standalone JavaScript applications are becoming more and more popular. In this environment, the user downloads a self-contained application, often containing a GUI developed in HTML and executes it locally. The main advantages are the portability of the code across operating systems and the reuse of web-related skills among developers. However, there are a lot of challenges. First, JavaScript is single-threaded and therefore cannot easily take advantage of multiple CPU cores. Second, since JavaScript is traditionally deployed as source code, an important question is how to protect the intellectual property when deploying to clients. Third, the lack of encapsulation in the language may cause security issues when running the code outside of the browser. Even though all these issues make the topic attractive for a program analysis researcher, we are not aware of any work that considers this environment, with the exception of a static analysis applied to desktop widgets [Guarnieri and Livshits 2009].

In addition to challenges arising on particular platforms, future work should consider code running on multiple platforms. For example, developers tend to reuse code across different JavaScript platforms,[15] ignoring the security particularities of each of them.

Interaction with Non-JavaScript Code. Many dynamic (and also static) program analyses ignore, or only partially consider, the behavior of code not implemented in JavaScript, such as native implementations of built-in APIs. The challenge here is to obtain a model of these parts of a JavaScript program. Since creating such a model is costly and likely to become invalid when the execution environment evolves, future work could automatically infer a model that describes the behavior of non-JavaScript code in terms of JavaScript code or some other behavior specification.

Evolution of the JavaScript Language. The evolution of the JavaScript language and its built-in APIs impose the challenge of adapting existing programs to novel language features and APIs. Since the dynamic nature of the language often requires (support by) dynamic analysis, we see potential for analyses that identify opportunities for adapting a given program to novel language features. As a very first step in this direction, JITProf [Gong et al. 2015a] (Section 5.2) suggests to use typed arrays, which where introduced in ECMAScript 2015, for improved performance. Another promising direction is related to optional type annotations understood by static checkers for JavaScript, such as Flow[16] and Closure[17], or introduced in variants of the JavaScript language, such as TypeScript [Typ 2016]. Future work should investigate how to integrate type information into dynamic analyses and test generators. For example, type annotations could guide test generators toward when picking input values, similar to type-guided test generation approaches for Java [Csallner and Smaragdakis 2004; Pacheco et al. 2007].

---

[14]According to http://w3techs.com/technologies/overview/programming_language/all, JavaScript is one of the top ten languages used for server-side development.

[15]For example, *browserify* is a popular tool enabling this kind of reuse.

[16]https://flowtype.org/

[17]https://developers.google.com/closure/compiler/

10.4. Support for Additional Developer Tasks

Sections 3 to 6 cover existing work that supports software developers during diverse activities, such as bug detection, optimization of bottlenecks, and increasing the security of a program. Beyond these tasks, there are several common developer activities that are currently not or only partially supported by dynamic analysis and test generation techniques. One of them is automated program repair, i.e., the process of finding a code change that avoids some misbehavior. While program repair is an active field of research, there is little work for JavaScript (Section 3.1). We envision future work in this area, e.g., on dynamic analyses that identify potential repairs and on test generation techniques that validate whether a potential repair indeed fixes the problem without affecting the remaining program behavior. Another developer activity that is currently insufficiently addressed by dynamic analysis is refactoring. Even though refactoring is traditionally a subject of static analysis, the dynamic nature of JavaScript suggests that (at least partly) dynamic approaches could effectively support developers in adapting their code. A major challenge is to ensure that refactorings are guaranteed, or at least very likely, to be semantics-preserving, which is inherently impossible to show with a purely dynamic analysis.

10.5. Guidance from Empirical Evidence

Several of the empirical studies covered in Section 7 have uncovered problems and opportunities addressed in subsequent research. We see several opportunities for future empirical studies, both in terms of topics to cover and in terms of code bases to analyze. As almost all existing studies focus on client-side web application code, one promising direction is to also study the properties of code running on other platforms, such as server-side web applications, mobile applications, and browser extensions. Another direction is to investigate the usage of yet unstudied language features and APIs, such as features introduced in HTML 5 and ECMAScript 6. An understudied aspect of JavaScript usage is how JavaScript code bases evolve over time, which one could study, e.g., using public code repositories or by repeatedly analyzing a set of web sites over a period of time. Finally, given the fast pace at which JavaScript and the web evolves as a whole, it will be interesting to reproduce some of the earlier studies to check to what extent their results have changed over time.

In addition to studying programs and other artifacts of the development process, there is a need for empirical studies of the usability of analysis tools for JavaScript. Understanding why developers use, or do not use, existing analysis tools will help improve future tools and eventually increase the impact of the research efforts surveyed in this article.

10.6. Reusable Research Infrastructure and Benchmarks

Given the large number of analyses that track dynamic dependencies (Section 6) and the fact that most of them are implemented from scratch, there is an opportunity for future work on a common framework for tracking dependencies. Another promising research direction is to develop a JavaScript virtual machine (VM) specifically designed with ease of extensions in mind. Due to the complexity of JavaScript and its optimizations, production VMs have complex implementations, making it non-trivial to modify them and to maintain modifications (Section 9). For comparison, the Java community has seen various improvements of runtime compilation and optimization techniques, many of which have been (initially) implemented in the Jikes research VM [Alpern et al. 2005]. In addition to providing an environment for implementing novel optimizations, a JavaScript research VM could also serve as a basis for implementing dynamic analyses. An alternative direction worth exploring is to build upon existing tools used

in industry, such as Selenium WebDriver,[18] that make it possible to programmatically control and monitor executions. It would be useful for the research community if the WebDriver API were extended to support more deep control and monitoring of the execution, for example, of event scheduling.

The large amount of competing approaches surveyed in this article provides an opportunity for novel benchmarks and tools to create such benchmarks. Currently, many papers are evaluated with an ad-hoc selection of programs and problems, making it difficult to understand the strength and weaknesses of related approaches. For example, it would be desirable to have a curated collection of JavaScript correctness bugs, e.g., similar to the Defects4J database for Java [Just et al. 2014]. Furthermore, it is necessary to consider the development of new or the adaptation of current benchmarks to support new language features and constructs. Many of the ECMAScript 6 and 7 features are already available in browsers, however, up to date, we are not aware of official benchmarks that demonstrate the performance of these features in JavaScript engines.

Is it important to note that are already steps toward reusable research infrastructures and benchmarks for JavaScript, such as Jalangi [Sen et al. 2013], which is the basis of various dynamic analyses, Crawljax [Mesbah et al. 2012], on which several test generators build, the performance benchmark creator JSBench [Richards et al. 2011a], and work by Selakovic and Pradel [2016], which provides reproducible versions of a large set of real-world performance issues.

11. CONCLUDING REMARKS

This article provides a survey of dynamic analysis and test generation techniques for JavaScript. As JavaScript is only 20 years old and has become popular for complex applications only within the last 10 years, the field of research is relatively young. Despite its young age, the field has already seen large amounts of work. By summarizing and comparing existing approaches in a structured way, we enable interested outsiders to quickly get an overview of the field. We conclude that the current state of the art successfully addresses the most common software goals – correctness, reliability, security, privacy, and performance, as well as the meta-goal of developer productivity.

Despite all the advances in analyzing JavaScript programs, various research challenges remain to be addressed. These include improved support for code refactoring and program repair, analyses targeted at emerging execution platforms and usage scenarios of JavaScript, and combinations of test generation and dynamic analysis. A particularly important goal for future work should be to consolidate existing research results into reusable research infrastructures that enable future research to avoid re-inventing the wheel. Moreover, to bridge the current gap between ideas presented in research and ideas picked up by industry, future work should strive for analysis tools that are easily usable by regular developers. Given the continuously increasing popularity of JavaScript, we expect further progress in dynamic analysis and test generation for JavaScript and hope that this article helps guiding it in fruitful directions.

REFERENCES

2016. TypeScript Language Specification, Version 1.8. (January 2016).

Hiralal Agrawal and Joseph Robert Horgan. 1990. Dynamic Program Slicing. In *PLDI*. 246–256.

Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2015. Hybrid DOM-Sensitive Change Impact Analysis for JavaScript. In *ECOOP*, Vol. 37. 321–345.

Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2016. Understanding asynchronous interactions in full-stack JavaScript. In *ICSE*. 1169–1180.

---

[18]http://www.seleniumhq.org/

Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2014. Understanding JavaScript event-based interactions. In *ICSE*. 367–377.

Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria A. Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark F. Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal* 44, 2 (2005), 399–418.

Esben Andreasen and Anders Møller. 2014. Determinacy in static analysis for jQuery. In *OOPSLA*. 17–31.

Silviu Andrica and George Candea. 2011. WaRR: A tool for high-fidelity web application record and replay. In *DSN*. 403–410.

Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A framework for automated testing of JavaScript web applications. In *ICSE*. 571–580.

Thomas H. Austin and Cormac Flanagan. 2010. Permissive dynamic information flow analysis. In *PLAS*. 3.

Thomas H. Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In *POPL*. 165–178.

John Aycock. 2003. A brief history of just-in-time. *ACM Comput. Surv.* 35, 2 (2003), 97–113.

Nathaniel Ayewah and William Pugh. 2010. The Google FindBugs fixit. In *ISSTA*. 241–252.

Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2014. Dompletion: DOM-aware JavaScript code completion. In *ASE*. 43–54.

Zahra Behfarshad and Ali Mesbah. 2013. Hidden-Web Induced by Client-Side Scripting: An Empirical Study. In *ICWE*, Vol. 7977. 52–67.

Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.

Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. 2014. Information Flow Control in WebKit's JavaScript Bytecode. In *POST*, Vol. 8414. 159–178.

Arnar Birgisson, Daniel Hedin, and Andrei Sabelfeld. 2012. Boosting the Permissiveness of Dynamic Information-Flow Tracking by Testing. In *ESORICS*, Vol. 7459. 55–72.

Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. 2013. Interactive record/replay for web application debugging. In *UIST*. 473–484.

Brian Burg, Andrew J. Ko, and Michael D. Ernst. 2015. Explaining Visual Changes in Web Interfaces. In *UIST*. 259–268.

Shauvik Roy Choudhary. 2014. Cross-platform testing and maintenance of web and mobile applications. In *ICSE*. 642–645.

Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2012. CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications. In *ICST*. 171–180.

Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2013. X-PERT: accurate identification of cross-browser issues in web applications. In *ICSE*. 702–711.

Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2014a. Cross-platform feature matching for web applications. In *ISSTA*. 82–92.

Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2014b. X-PERT: a web application testing tool for cross-browser inconsistency detection. In *ISSTA*. 417–420.

Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. 2010a. A cross-browser web application testing tool. In *ICSM*. 1–6.

Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. 2010b. WEBDIFF: Automated identification of cross-browser issues in web applications. In *ICSM*. 1–10.

Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, and Coen De Roover. 2016. Linvail: A General-Purpose Platform for Shadow Execution of JavaScript. In *SANER*. 260–270.

Andrey Chudnov and David A. Naumann. 2015. Inlined Information Flow Monitoring for JavaScript. In *CCS*. 629–643.

Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. 2009. Staged information flow for JavaScript. In *PLDI*. 50–62.

Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: an automatic robustness tester for Java. *Softw., Pract. Exper.* 34, 11 (2004), 1025–1050.

Igor Rafael de Assis Costa, Péricles Rafael Oliveira Alves, Henrique Nazaré Santos, and Fernando Magno Quintão Pereira. 2013. Just-in-time value specialization. In *CGO*. 29:1–29:11.

Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-execution. In *S&P*. 109–124.

Mohan Dhawan and Vinod Ganapathy. 2009. Analyzing Information Flow in JavaScript-Based Browser Extensions. In *ACSAC*. 382–391.

Monika Dhok, Murali Krishna Ramanathan, and Nishant Sinha. 2016. Type-aware concolic testing of JavaScript programs. In *ICSE*. 168–179.

Serdar Dogan, Aysu Betin-Can, and Vahid Garousi. 2014. Web application testing: A systematic literature review. *Journal of Systems and Software* 91 (2014), 174–201.

ECMA. 2016. Standard ECMA-262, ECMAScript Language Specification, 7th Edition. (June 2016).

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *SCP* 69, 1-3 (2007), 35–45.

Amin Milani Fard and Ali Mesbah. 2013. JSNOSE: Detecting JavaScript Code Smells. In *SCAM*. 116–125.

Amin Milani Fard, Ali Mesbah, and Eric Wohlstadter. 2015. Generating Fixtures for JavaScript Unit Testing. In *ASE*. 190–200.

Amin Milani Fard, Mehdi MirzaAghaei, and Ali Mesbah. 2014. Leveraging existing tests in automated test generation for web applications. In *ASE*. 67–78.

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 *PLDI*. 465–478.

Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. 2015. Don't Call Us, We'll Call You: Characterizing Callbacks in JavaScript. In *ESEM*. 247–256.

Vahid Garousi, Ali Mesbah, Aysu Betin-Can, and Shabnam Mirshokraie. 2013. A systematic mapping study of web application testing. *Information & Software Technology* 55, 8 (2013), 1374–1396.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *PLDI*. 213–223.

Liang Gong, Michael Pradel, and Koushik Sen. 2015a. JITProf: pinpointing JIT-unfriendly JavaScript code. In *ESEC/FSE*. 357–368.

Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015b. DLint: dynamically checking bad coding practices in JavaScript. In *ISSTA*. 94–105.

Mark Grechanik, Qing Xie, and Chen Fu. 2009. Maintaining and evolving GUI-directed test scripts. In *ICSE*. 408–418.

Salvatore Guarnieri and V. Benjamin Livshits. 2009. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium*. 151–168.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2011. Typing Local Control and State Using Flow Analysis. In *ESOP*, Vol. 6602. 256–275.

Brian Hackett and Shu-yu Guo. 2012. Fast and precise hybrid type inference for JavaScript. In *PLDI*. 239–250.

Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: tracking information flow in JavaScript and its APIs. In *SAC*. 1663–1671.

Daniel Hedin and Andrei Sabelfeld. 2012. Information-Flow Security for a Core of JavaScript. In *CSF*. 3–18.

Phillip Heidegger and Peter Thiemann. 2012. JSConTest: Contract-Driven Testing and Path Effect Inference for JavaScript. *Journal of Object Technology* 11, 1 (2012), 1–29.

Joshua Hibschman and Haoqi Zhang. 2015. Unravel: Rapid Web Application Reverse Engineering via Interaction Recording, Source Tracing, and Library Detection. In *UIST*. 270–279.

Shin Hong, Yongbae Park, and Moonzoo Kim. 2014. Detecting Concurrency Errors in Client-Side Java Script Web Applications. In *ICST*. 61–70.

James Ide, Rastislav Bodik, and Doug Kimelman. 2009. Concurrency concerns in rich Internet applications. In *ECEC*.

Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. An empirical study of privacy-violating information flows in JavaScript web applications. In *CCS*. 270–283.

Casper Svenning Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin T. Vechev. 2015a. Stateless model checking of event-driven applications. In *OOPSLA*. 57–73.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *SAS*, Vol. 5673. 238–255.

Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. 2015b. MemInsight: platform-independent memory debugging for JavaScript. In *ESEC/FSE*. 345–356.

Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. 2014. Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation. In *CCS*. 66–77.

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*. 437–440.

Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and Chung-chieh Shan. 2012. An Analysis of the Mozilla Jetpack Extension Framework. In *ECOOP*, Vol. 7313. 333–355.

Madhukar N. Kedlaya, Behnam Robatmili, and Ben Hardekopf. 2015. Server-side type profiling for optimizing client-side JavaScript engines. In *DLS*. 140–153.

Haruka Kikuchi, Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. 2008. JavaScript Instrumentation in Practice. In *APLAS*. 326–341.

Erick Lavoie, Bruno Dufour, and Marc Feeley. 2014. Portable and Efficient Run-time Monitoring of JavaScript Applications Using Virtual Machine Layering. In *ECOOP*, Vol. 8586. 541–566.

Sungho Lee, Julian Dolby, and Sukyoung Ryu. 2016. HybriDroid: static analysis framework for Android hybrid applications. In *ASE*. 250–261.

Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *CCS*. 1193–1204.

Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2014. Reducing Web Test Cases Aging by Means of Robust XPath Locators. In *ISSRE*. 449–454.

Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014a. SymJS: automatic symbolic testing of JavaScript web applications. In *FSE*. 449–459.

Yuan-Fang Li, Paramjit K. Das, and David L. Dowe. 2014b. Two decades of Web application testing - A survey of recent advances. *Inf. Syst.* 43 (2014), 20–54.

Magnus Madsen, Frank Tip, Esben Andreasen, Koushik Sen, and Anders Møller. 2016. Feedback-directed instrumentation for deployed JavaScript applications. In *ICSE*. 899–910.

Josip Maras, Jan Carlson, and Ivica Crnkovic. 2011. Client-side web application slicing. In *ASE*. 504–507.

Josip Maras, Jan Carlson, and Ivica Crnkovic. 2012. Extracting client-side web application code. In *WWW*. 819–828.

Nick Matthijssen, Andy Zaidman, Margaret-Anne D. Storey, R. Ian Bull, and Arie van Deursen. 2010. Connecting Traces: Understanding Client-Server Interactions in Ajax Applications. In *ICPC*. 216–225.

Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. 2012. Eval begone!: semi-automated removal of eval from JavaScript programs. In *OOPSLA*. 607–620.

Ali Mesbah. 2015. Advances in Testing JavaScript-Based Web Applications. *Advances in Computers* 97 (2015), 201–235.

Ali Mesbah and Mukul R. Prasad. 2011. Automated cross-browser compatibility testing. In *ICSE*. 561–570.

Ali Mesbah, Arie van Deursen, and Stefan Lenselink. 2012. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. *TWEB* 6, 1 (2012), 3.

James W. Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for JavaScript Applications. In *NSDI*. 159–174.

Edward Miller and William E Howden. 1981. *Software testing & validation techniques*.

Shabnam Mirshokraie and Ali Mesbah. 2012. JSART: JavaScript Assertion-Based Regression Testing. In *ICWE*, Vol. 7387. 238–252.

Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2013. PYTHIA: Generating test cases with oracles for JavaScript applications. In *ASE*. 610–615.

Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. JSEFT: Automated JavaScript Unit Test Generation. In *ICST*. 1–10.

Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2015. Detecting JavaScript races that matter. In *ESEC/FSE*. 381–392.

Alex Nederlof, Ali Mesbah, and Arie van Deursen. 2014. Software engineering for the web: the state of the practice. In *ICSE*. 4–13.

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 *CCS*. 736–747.

Frolin S. Ocariza Jr., Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2013. An Empirical Study of Client-Side JavaScript Bugs. In *ESEM*. 55–64.

Frolin S. Ocariza Jr., Karthik Pattabiraman, and Ali Mesbah. 2012. AutoFLox: An Automatic Fault Localizer for Client-Side JavaScript. In *ICST*. 31–40.

Frolin S. Ocariza Jr., Karthik Pattabiraman, and Ali Mesbah. 2014. Vejovis: suggesting fixes for JavaScript faults. In *ICSE*. 837–847.

Frolin S. Ocariza Jr., Karthik Pattabiraman, and Benjamin G. Zorn. 2011. JavaScript Errors in the Wild: An Empirical Study. In *ISSRE*. 100–109.

Stephen Oney and Brad A. Myers. 2009. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *VL/HCC*. 105–108.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *ICSE*. 75–84.

Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *CGO*. 2–11.

Boris Petrov, Martin T. Vechev, Manu Sridharan, and Julian Dolby. 2012. Race detection for web applications. In *PLDI*. 251–262.

Michael Pradel, Parker Schuh, George C. Necula, and Koushik Sen. 2014. EventBreak: analyzing the responsiveness of user interfaces through performance-guided test generation. In *OOPSLA*. 33–47.

Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript. In *ICSE*. 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 *ECOOP*, Vol. 37. 519–541.

Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. 2010. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *WebApps*.

Veselin Raychev, Martin T. Vechev, and Manu Sridharan. 2013. Effective race detection for event-driven programs. In *OOPSLA*. 151–166.

Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. 2011a. Automated construction of JavaScript benchmarks. In *OOPSLA*. 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 *ECOOP*, Vol. 6813. 52–78.

Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*. 1–12.

Philippe De Ryck, Maarten Decat, Lieven Desmet, Frank Piessens, and Wouter Joosen. 2010. Security of Web Mashups: A Survey. In *NordSec*. 223–238.

Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.

Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *ICSE*. 598–608.

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010a. A Symbolic Execution Framework for JavaScript. In *S&P*. 513–528.

Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. 2010b. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS*.

Marija Selakovic and Michael Pradel. 2016. Performance issues and optimizations in JavaScript: an empirical study. In *ICSE*. 61–72.

Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE*. 488–498.

Koushik Sen, George C. Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: multi-path symbolic execution using value summaries. In *FSE*. 842–853.

Charles R. Severance. 2012. JavaScript: Designing a Language in 10 Days. *IEEE Computer* 45, 2 (2012), 7–8.

Sooel Son and Vitaly Shmatikov. 2013. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *NDSS*.

Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP*, Vol. 7313. 435–458.

Vincent St-Amour and Shu-yu Guo. 2015. Optimization Coaching for JavaScript (Artifact). *DARTS* 1, 1 (2015), 05:1–05:2.

Hallvord Reiar Michaelsen Steen. 2009. websites playing timing roulette. https://hallvors.wordpress.com/2009/03/07/websites-playing-timing-roulette/. (March 2009).

Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. 2014. Precise Client-side Protection against DOM-based Cross-Site Scripting. In *USENIX Security Symposium*. 655–670.

Hideo Tanida, Tadahiro Uehara, Guodong Li, and Indradeep Ghosh. 2015. Automated Unit Testing of JavaScript Code through Symbolic Executor SymJS. (2015).

Peter Thiemann. 2005. Towards a Type System for Analyzing JavaScript Programs. In *ESOP*, Vol. 3444. 408–422.

Omer Tripp, Pietro Ferrara, and Marco Pistoia. 2014. Hybrid security analysis of web JavaScript code via dynamic partial evaluation. In *ISSTA*. 49–59.

Shiyi Wei and Barbara G. Ryder. 2013. Practical blended taint analysis for JavaScript. In *ISSTA*. 336–346.

Shiyi Wei, Franceska Xhakaj, and Barbara G. Ryder. 2016. Empirical study of the dynamic behavior of JavaScript objects. *Softw., Pract. Exper.* 46, 7 (2016), 867–889.

Xiao Xiao, Shi Han, Charles Zhang, and Dongmei Zhang. 2015. Uncovering JavaScript Performance Code Smells Relevant to Type Mutations. In *APLAS*, Vol. 9458. 335–355.

Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. 2007. JavaScript instrumentation for browser security. In *POPL*. 237–249.

Chuan Yue and Haining Wang. 2009. Characterizing insecure JavaScript practices on the web. In *WWW*. 961–970.

Sai Zhang, Hao Lü, and Michael D. Ernst. 2013. Automatically repairing broken workflows for evolving GUI applications. In *ISSTA*. 45–55.

Yunhui Zheng, Tao Bao, and Xiangyu Zhang. 2011. Statically locating web application bugs caused by asynchronous calls. In *WWW*. 805–814.