2015 30th IEEE/ACM International Conference on Automated Software Engineering

Static Analysis of JavaScript Web Applications in the Wild via Practical DOM Modeling

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

Sooncheol Won
KAIST
wonsch@kaist.ac.kr

Joonho Jin
KAIST
myfriend12@kaist.ac.kr

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

***Abstract*—We present SAFE$_{\text{WApp}}$, an open-source static analysis framework for JavaScript web applications. It provides a faithful (partial) model of web application execution environments of various browsers, based on empirical data from the main web pages of the 9,465 most popular websites. A main feature of SAFE$_{\text{WApp}}$ is the configurability of DOM tree abstraction levels to allow users to adjust a trade-off between analysis performance and precision depending on their applications. We evaluate SAFE$_{\text{WApp}}$ on the 5 most popular JavaScript libraries and the main web pages of the 10 most popular websites in terms of analysis performance, precision, and modeling coverage. Additionally, as an application of SAFE$_{\text{WApp}}$, we build a bug detector for JavaScript web applications that uses static analysis results from SAFE$_{\text{WApp}}$. Our bug detector found previously undiscovered bugs including ones from wikipedia.org and amazon.com.**

I. Introduction

JavaScript was originally developed as a simple scripting language to develop interactive and dynamic web pages, but now it is used to construct large web applications with the evolution of web contents. Although JavaScript is the most dominant programming language on the web [1], its quirky features make it difficult to understand behaviors of complex JavaScript programs. Many researchers have developed various program analysis techniques to alleviate this problem, but many of them have focused on analyzing stand-alone JavaScript programs [28], [4], [13], which is not enough for analyzing real-world web applications.

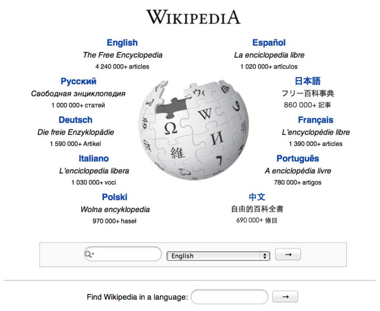
To analyze JavaScript web applications in the wild, modeling browser environments is critical since JavaScript programs often reside in web pages and interact with users and browsers via events. However, one of the main challenges is that no single standard document describes browser environments formally. While some commonly-used specifications are available online [6], [7], their descriptions are huge, informal, and incomplete and modeling all of them manually is tedious, time consuming, and error prone. Furthermore, non-standard and inconsistent behaviors of various browsers complicate modeling. While previous approaches presented partial models of browser environments selected from several documents, they are not applicable to real-world web applications because they support only subsets of JavaScript or limited models of web documents [4], [8], [9], [10], [11], [12].

In this paper, we present SAFE$_{\text{WApp}}$, an open-source framework that statically analyzes JavaScript web applications and show how to extend the existing static analyzer SAFE [13] to support an extensive model of browser environments. By doing so, SAFE$_{\text{WApp}}$ extends the applicability of any client applications that are built on top of SAFE from stand-alone JavaScript programs to web applications. To implement a practical model of browser environments, we investigated 9,465 popular websites and summarized uses of native JavaScript objects and APIs provided by most major web browsers. Based on extensive empirical data, we design a model of browser environments. A main feature of SAFE$_{\text{WApp}}$ is the configurability of DOM tree abstraction levels in three ways, two from the literature [11], [12] and one new, which maintains precise tree structures of web documents. We show how DOM tree abstraction levels affect analysis performance and precision with the analysis results of the 5 most popular JavaScript libraries and the main web pages of the 10 most popular websites. Furthermore, we implement a bug detector capable of detecting language-level bugs as a client application of SAFE$_{\text{WApp}}$ and show that it detects previously undiscovered bugs in the main web pages of popular websites. Note that our approach is not tied to its base analyzer, SAFE, and thus it is applicable to other static analyzers for JavaScript programs.

The contributions of this paper include the followings:

- We present *empirical data from the 9,465 most popular websites* to show the usage of native objects and APIs that browsers support. This data will be helpful to model browser environments practically.
- We explain how to extend a static analyzer that supports only stand-alone JavaScript programs to the one that supports web applications with a model of browser environments practically. This approach is applicable to other static JavaScript analyzers.
- We present SAFE$_{\text{WApp}}$, *a framework that statically analyzes* real-world JavaScript web applications. It provides *configurable DOM tree abstraction models* in three levels, two existing ones [11], [12] and a new one that captures DOM tree structures of web documents precisely. We present experimental results about the effects of DOM tree abstraction levels on analysis results of real-world web applications such as JavaScript libraries and websites.
- To show the usefulness of SAFE$_{\text{WApp}}$, we implement a bug detector as an application of SAFE$_{\text{WApp}}$ and evaluate it on the main web pages of the 10 most popular websites. We *found previously undiscovered bugs* in `wikipedia.org` and `amazon.com`.
- We make SAFE$_{\text{WApp}}$ and the bug detector *publicly available* [14].

978-1-5090-0025-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ASE.2015.27



```
function setLang(lang) {
  var uiLang = navigator.language ||
               navigator.userLanguage,
      date = new Date();
  if (uiLang.match(/^\w+/) === lang) {
     date.setTime(date.getTime() - 1);
  } else {
     date.setFullYear(date.getFullYear() + 1);
  }
  document.cookie = "searchLang=" + lang + ";expires=" +
                    date.toUTCString() + ";domain=" +
                    location.host + ";";
}
```

Fig. 1: Main web page of `wikipedia.org`, along with some JavaScript code in the page (before fixing the bug)

II. Motivating Examples

This section presents two examples motivating design decisions of $\text{SAFE}_{\text{WApp}}$. The first example is the bug we found from `wikipedia.org`, which requires analysis of interactions between JavaScript and DOM APIs. The second example requires precise DOM modeling for precise analysis.

First, the JavaScript code of the `setLang` function in Figure 1 is an excerpt from the main web page of Wikipedia also shown in the Figure. A user can select a language in which search results are displayed via the selection button next to the search box; in Figure 1, the selected language is 'English.' When the user changes the selected language to another one via the button, it calls the `setLang` function. The intended semantics of `setLang` is as follows. If the selected language is different from the default language of the browser in use, the function saves the information in a cookie on the user's computer and keeps it for a year; otherwise, it simply discards cookies, if any. When the user visits Wikipedia next time, if the cookie keeps the previously-selected language, the site uses it to set the default language in the selection button. Otherwise, it sets the default language to the browser's default one.

However, we found that a bug in `setLang` produces an unintended behavior. When the `if` statement compares the browser's default language "`uiLang.match(/^\w+/)`" with the selected language `lang`, the comparison always evaluates to false! Note that the result of the `match` function is either an object or `null`, while the value of `lang` is always a string. Because the strict equals operator (===) always returns false if the types of two operands are different, the conditional expression always evaluates to false, which always sets the expiration date of the cookie to one year later. Consequently, whenever `setLang` is called, the time in `date` sets to one year later of the current time, and the cookie never expires immediately leaving the information on the user's computer.

We reported the bug to the Wikipedia developers, and they confirmed the bug and fixed it right away [**?**]. We can fix the bug by using the equals operator == instead of ===, because == implicitly converts two operands of different types. We found that Wikipedia uses the strict mode recommended by ECMAScript in their JavaScript code to enforce more checks, but the example shows that the checks are not enough to catch such a bug while a bug detector using $\text{SAFE}_{\text{WApp}}$ can detect it. Note that the modeling of both the event system and browser environments is necessary to catch the bug precisely. Without considering the event system, the `setLang` call is unreachable and without the precise type information of browser languages (`navigator.language`), we cannot detect the conditional expression as a definite bug.

The second is the following pattern in web documents [15]:

```
var canvas = document.querySelector("#leftcol .logo")
```

While the `document.querySelector` call returns an object of `HTMLCanvasElement` by searching the DOM tree, common approaches [11], [12] that model the DOM tree imprecisely fail to provide precise analysis results; they simply return any HTML element, which includes other elements as well resulting in imprecise analysis results. For more precise analysis results, more precise modeling of DOM is necessary [3]. Hence, $\text{SAFE}_{\text{WApp}}$ provides not only existing DOM tree abstraction models in the literature but also a new precise model of DOM trees; with the new model, it can return a single `HTMLCanvasElement` object by searching an abstract DOM tree as a result of the `document.querySelector` call in an HTML document with the `<canvas>` tag element. In Section IV, we show that such APIs for searching DOM trees are frequently used in real-world web applications, and in Section VII, we show how DOM tree abstraction levels affect static analysis results of JavaScript web applications.

III. JavaScript in Web Applications

We explain the execution model of JavaScript web applications and discuss challenges in analyzing them statically.

A. *Execution of JavaScript Web Applications*

A JavaScript web application runs in a web browser and users can interact with it through user events such as mouse and keyboard events. Figure 2 illustrates an execution of a simple JavaScript web application, which first shows a picture of eggs and then changes it to a picture of a chicken when a user clicks it. The source of the web application is an HTML document consisting of a series of HTML tags. JavaScript code usually resides inside `<script>` tags, either inlined as in Figure 2 or imported from an external source as follows:

```
<script type="text/javascript"
        src="sourcepath/sourcefile.js">
```

Such JavaScript code can access the enclosing HTML document and change its structure, contents, and display.

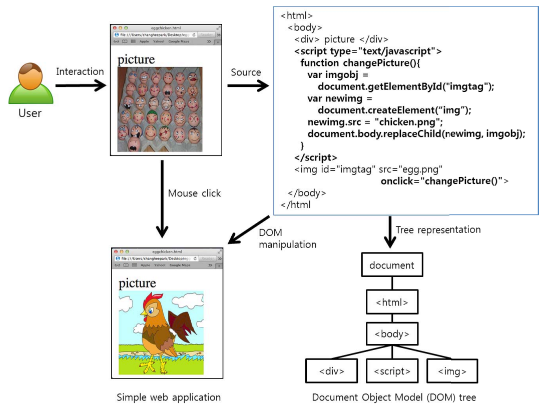

```
<html>
  <body>
    <div> picture </div>
    <script type="text/javascript">
     function changePicture(){
       var imgobj =
          document.getElementById("imgtag");
       var newimg =
          document.createElement("img");
       newimg.src = "chicken.png";
       document.body.replaceChild(newimg, imgobj);
     }
    </script>
    <img id="imgtag" src="egg.png"
                          onclick="changePicture()">
  </body>
</html
```

Fig. 2: Execution of a sample JavaScript web application

To allow JavaScript code to manipulate HTML documents, web browsers provide *the Document Object Model (DOM)*, a tree representation of HTML documents. A node in DOM usually represents an HTML tag and is treated as a JavaScript object with links to its parent, children, and sibling nodes. The tree diagram in Figure 2 shows a DOM tree for the HTML document in the Figure. Note that actual DOM trees created by web browsers are often more complex than the one in the figure. The `changePicture` function inside the `<script>` tag of the HTML document in Figure 2 shows an example where JavaScript code changes the contents of its enclosing HTML document by manipulating DOM. It first gets an `HTMLImageElement` object representing the `<img>` tag in the HTML document by calling `document.getElementById` and it assigns the object to `imgobj`. Then, it creates a new `HTMLImageElement` object by calling `document.createElement` and designates the source of the new image as the path to a new picture. Finally, it changes the structure of DOM by replacing a child of the body element `imgobj` with the new object `newimg`.

Functions like `getElementById` and `createElement` are not described by the ECMAScript specification. They are like *APIs* that browsers provide to enable JavaScript code to access browser environments such as DOM trees. *DOM objects* returned by such browser APIs correspond to HTML elements. They share some common properties such as links to their parents and children nodes, but some DOM objects have their own properties: for example, both objects for `HTMLImageElement` and `HTMLDivElement` have the `parentNode` property in common, but only the objects for `HTMLImageElement` have the `src` property. Browsers usually implement DOM objects and browser APIs in native code for performance reasons.

Finally, most web applications run with a variety of events from user interface events like keyboard and mouse events to browser behavioral events like load and change events. The event system enables developers to catch event triggers and to define actions to handle events by registering event handler functions. For example, `changePicture` is an event handler function registered to the `onclick` attribute of the `<img>` tag with the picture of eggs. When a user clicks the picture, the event system calls `changePicture` and changes the picture to the picture of a chicken.

B. Static Analysis Challenges

Precisely and correctly analyzing JavaScript web applications requires modeling of browser environments including DOM, browser APIs, and the event system in a base analyzer. Without the modeling, for example, when SAFE supporting only stand-alone JavaScript analyzes a simple web application in Figure 2, it does not analyze the `changePicture` call because the function is unreachable without an event system; even if it can analyze the function call, it signals a run-time exception at the first line of the function body with undeclared variable access to `document` because `document` is an accessor that browsers provide for the root element of the DOM tree; then it terminates the analysis because the rest of source parts are unreachable due to the run-time exception.

However, such modeling involves several challenges. First, because no single standard specification describes browser environments, it is not clear what to model among thousands of properties related to DOM objects and browser APIs. While some commonly-used specifications [6], [7] about browser environments are available, they are huge, informal, and incomplete. Moreover, modeling DOM objects only in the documents may be insufficient because many browsers support non-standard features: for example, while the `Screen` object is not a part of any specifications, we found that more than 70% of the 10,000 most popular websites use it in their web pages. In addition, incompatibility between browsers makes it even more difficult to select modeling targets: for example, while Internet Explorer provides `attachEvent` to register event handlers, neither of Safari and Chrome provides it. Instead of modeling only subsets of certain documents, we model DOM objects and browser APIs based on our extensive empirical study. We present the empirical data in the next section.

Second, because execution of JavaScript code in HTML documents may interleave with parsing of HTML documents, only a partial structure of a DOM tree may be visible in the JavaScript code of a certain `<script>` tag and events may occur while parsing HTML documents. To precisely analyze such an execution, we may need to incorporate HTML parsing with static analysis, which requires a meta-circular analyzer. Instead, we simplify the execution model; we assume that all scripts are executed after completion of HTML parsing and all the events happen after execution of the top-level code.

IV. Practical DOM Modeling

This section presents empirical data to identify frequently used DOM objects and browser APIs. The purpose of this empirical study is to lessen the burden of modeling efforts by suggesting modeling priorities on DOM properties and DOM tree abstraction levels. While modeling mechanism itself is not a main focus of our work, automatic modeling mechanism as in SAFE$_{\text{WAPI}}$ [2] can generate modeling from API specifications.

A. Method

We investigated what DOM-related fields and browser APIs are commonly used in real-world JavaScript applications. For this, we instrumented WebKit[1], an open-source web browser engine used in Safari and Chrome, to identify DOM-related

[1]http://www.webkit.org

TABLE I: Usage of fields/APIs defined in DOM specifications

Spec.	Fields		APIs	
	Def.	No use	Def.	No use
3 Core	120	67 (55.8%)	88	51 (58%)
2 HTML	288	126 (43.8%)	36	23 (63.9%)
2 Events	57	25 (43.9%)	14	4 (28.6%)
Total	465	218 (46.9%)	138	78 (56.5%)

TABLE II: Usage of fields/APIs in main pages of 9,465 sites

Type	> 1,000 sites	≥ 100 sites	≥ 10 sites	≥ 1 site
Field	148	325	949	1,719
API	52	100	162	269
Total	200	425	1,111	1,988

property accesses. Specifically, WebKit has two modules, JavaScriptCore and WebCore, which deal with JavaScript code execution and web page rendering, respectively. Since JavaScriptCore communicates with WebCore when in need to access DOM properties, we can identify all the accesses by observing the communications between two modules.

Using the instrumented WebKit, we made MiniBrowser, a simple browser provided by WebKit, navigate to 9,465 sites excluding inaccessible 535 sites from the 10,000 most popular websites according to Alexa[2] and stay for 1 minute per site just loading the main web pages. Whenever MiniBrowser visits a new website, the instrumented WebKit dumps various information such as the names of all DOM fields and APIs accessed in the main web page for 1 minute. We also built a tool to compute statistical data like the total number of accesses for each field or API and the number of websites where each field or API is used.

B. Results

Usage of the Fields and APIs Defined in Specifications. First, we demonstrate that it is not necessary to model all the fields and APIs defined in various DOM specifications. Table I summarizes the usage of the fields and APIs defined in 3 kinds of DOM specifications in the W3C community: DOM Level 3 Core, DOM Level 2 HTML, and DOM Level 2 Events, from top to bottom. For each specification, the columns of the table show, from left to right, the number of defined fields, the number of the fields never used in any site, the number of defined APIs, and the number of the APIs never used in any site. As the last row shows, about half of the defined fields and APIs were never used during execution of JavaScript code in the main web pages of 9,465 popular websites. Interestingly, even for the DOM Level 3 Core specification, which defines a core subset of web APIs, more than a half of the fields and APIs were never used. This result shows that most JavaScript web applications use only some subset of the DOM properties.

Modeling Priorities. While modeling all the properties defined in all DOM specifications would be most useful, it would be practically helpful to set priorities between the DOM properties so that one can incrementally model them in the order of importance. Table II summarizes the numbers of the fields and APIs used in the main web pages of the target websites. The second column shows that 148 fields and 52 APIs were found in more than 1,000 sites, and the fifth column shows that 1,719 fields and 269 APIs were found in more than or equal to 1 site, which means that the total number of DOM properties used in 9,465 websites are 1,988.

[2]http://www.alexa.com/topsites

TABLE III: 10 most frequently used DOM properties defined in the W3C DOM Level 3 Core specification

Rank	Interface	Field	Sites
1	`Document`	`documentElement`	8,116
2	`NodeList`	`length`	8,104
3	`Node`	`parentNode*`	8,008
4	`Node`	`nodeType`	7,718
5	`Node`	`firstChild*`	7,669
6	`Node`	`ownerDocument`	7,497
7	`Node`	`childNodes*`	7,354
8	`Node`	`nodeName`	7,297
9	`Node`	`lastChild*`	6,835
10	`Node`	`nextSibling*`	5,319
Rank	**Interface**	**API**	**Sites**
1	`Document`	`getElementById*`	7,441
2	`Document`	`createElement`	7,425
3	`Node`	`appendChild*`	7,393
4	`Document`	`getElementsByTagName*`	7,327
5	`Node`	`insertBefore*`	7,182
6	`Element`	`getElementsByTagName*`	7,134
7	`Element`	`getAttribute`	7,060
8	`Node`	`removeChild*`	7,018
9	`Element`	`setAttribute`	6,868
10	`Document`	`createComment`	6,427

While modeling all 1,988 properties found in 9,465 websites would be time-consuming and tedious, if we give a higher priority to those found in more than 1,000 sites, the modeling becomes manageable with 200 properties. After modeling them first, one can model other properties selectively depending on the target applications. For instance, we modeled about 800 properties using hand-written code; the model includes 196 among 200 properties found in more than 1,000 sites; we did not model 4 properties such as `HTMLDocument.write`, which require HTML parsing and possibly dynamic JavaScript source loading. We make the full list of the properties used in websites publicly available [14].

DOM Tree Modeling. The abstraction level of concrete DOM trees affects analysis precision and performance. For simple treatment of abstract objects, existing approaches abstract a DOM tree as either one single node object [11] or multiple objects to keep separate abstract objects only for different kinds of DOM elements [12]; both approaches give up maintaining the original tree structure at the initial state. While the approaches can always keep small fixed numbers of abstract objects for a DOM tree regardless of its original structure, they cannot provide precise analysis results for web applications manipulating the tree structure [3]. On the other hand, if a model maintains the DOM tree structure by abstracting each node as a single abstract object, the analyzer can provide more precise analysis results with the cost of more sophisticated modeling work and a large number of abstract objects depending on source HTML documents.

Observation on which fields and APIs are frequently used in real-world web applications can help to decide the abstraction level of concrete DOM trees. Table III shows the 10 most frequently used fields and APIs defined in DOM Level 3 Core with the numbers of the websites where they were found. The properties marked with $\star$ indicate that they either search or manipulate DOM trees. As the table shows, more than a half of the websites are using them, which implies that the

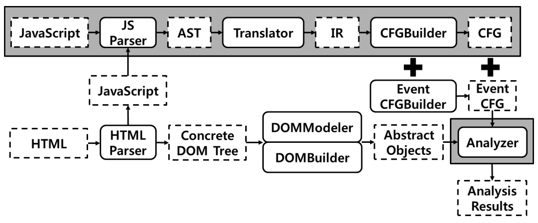


Fig. 3: Overall structure of the $\mathsf{SAFE}_{\mathsf{WApp}}$ framework

DOM-related properties are being used significantly in many web applications. Therefore, a crude abstraction that does not maintain DOM tree structures would lead to imprecise analysis results in web applications. Based on empirical data, our model provides a precise DOM tree abstraction model but we make it configurable to other two aforementioned existing abstraction models allowing users to choose an appropriate one depending on their applications. We provide more detailed explanation of the three DOM tree abstraction models in Section V-D.

V. $\mathsf{SAFE}_{\mathsf{WAPP}}$ FRAMEWORK

Now, we present $\mathsf{SAFE}_{\mathsf{WApp}}$ built on top of an open-source SAFE framework [16] as illustrated in Figure 3. Solid and dashed boxes represent modules and data, respectively, and arrows show flows of data between modules. The modules in the shaded boxes indicate the ones in SAFE and the others outside the shaded boxes are new additions.

A. *SAFE and Static Analyzer*

We give a brief introduction to SAFE and its analyzer in this section and refer the interested readers to its manual [17]. SAFE is an open-source analysis framework for stand-alone JavaScript programs. It transforms a JavaScript source program to an Abstract Syntax Tree (AST), an Intermediate Representation (IR) with simpler language constructs than AST, and a Control Flow Graph (CFG) to enable various analyses on different levels of representations. Its default static analyzer on CFGs supports flow-sensitive and context-sensitive analyses of JavaScript programs by faithfully modeling the semantics of ECMAScript 5 [18]. A SAFE analysis computes the following summary map for a program:

$$\hat{s} \in \widehat{\mathsf{S}} = \mathsf{Node} \times \mathsf{Context} \to \widehat{\mathsf{Heap}}$$

which maps a program point represented by a pair of a CFG node and a context to an over-approximate abstract heap information. A variable or an object property in an abstract heap, $\hat{h} \in \widehat{\mathsf{Heap}}$, maps to an abstract value $\hat{v}$ represented by a 6-tuple of lattice elements as follows:

$$\hat{v} \in \widehat{\mathsf{V}} = \widehat{\mathsf{Undef}} \times \widehat{\mathsf{Null}} \times \widehat{\mathsf{Bool}} \times \widehat{\mathsf{Number}} \times \widehat{\mathsf{String}} \times \wp(\widehat{\mathsf{Loc}})$$

where $\wp(\widehat{\mathsf{Loc}})$ is a finite set of abstract locations that map to abstract objects in abstract heaps and the others are simple abstract domains for primitive types, `Undefined`, `Null`, `Bool`, `Number`, and `String`. Their definitions are available in the SAFE manual [17]. For example, an abstract value $\hat{v}$ that may be `true` or `null` is represented as follows:

$$\hat{v} = \langle \bot_{\mathrm{Undef}}, \widehat{\mathsf{null}}, \widehat{\mathsf{true}}, \bot_{\mathrm{Number}}, \bot_{\mathrm{String}}, \varnothing \rangle.$$

With the domains, the default static analyzer performs sound and elaborate analyses on CFGs of JavaScript programs with the transfer function $\hat{F} \in \widehat{\mathsf{S}} \to \widehat{\mathsf{S}}$ to compute a final summary map $\hat{s}_{\mathsf{final}}$ from the following least fixpoint computation:

$$\hat{s}_{\mathsf{final}} = \mathsf{leastFix}\ \lambda\hat{s}.(\hat{s}_{\mathrm{I}} \sqcup_{\widehat{\mathsf{S}}} \hat{F}(\hat{s}))$$

where the initial summary map $\hat{s}_{\mathrm{I}}$ maps an initial program point to the initial heap $\hat{h}_{\mathsf{init}}$ and all the other program points to $\bot_{\mathrm{Heap}}$. The analysis starts with $\hat{s}_{\mathrm{I}}$ and updates the map until it reaches a fixpoint. The initial heap $\hat{h}_{\mathsf{init}}$ contains abstract value information of JavaScript built-in objects and functions. In addition, the analysis supports various analysis techniques such as object recency abstraction [19] (distinguishing recently allocated objects from summary objects at a program point), k-CFA [20] (distinguishing function calls with k length of call strings that represent call history), and loop-sensitivity [21] (distinguishing loop iterations during analysis using loop contexts that contain loop information) to improve analysis precision. The SAFE analysis supports such multiple analysis sensitivities by configuring Context accordingly. Note that $\mathsf{SAFE}_{\mathsf{WApp}}$ extends $\hat{h}_{\mathsf{init}}$ of the SAFE analyzer with more information about DOM trees and browser APIs so that it can analyze web applications. Without such information, when an input program calls browser APIs, for example, the SAFE analyzer wrongly identifies them as undefined function accesses or non-function calls making further analysis unsound.

B. *HTML Parser and Event System*

The first step to extend SAFE for stand-alone JavaScript programs to web applications is to provide a way to parse HTML documents. Because the WebKit HTML parser is not well structured or documented enough to provide necessary information about the HTML documents being parsed, $\mathsf{SAFE}_{\mathsf{WApp}}$ uses a combination of two third-party HTML parsers with different capabilities. $\mathsf{SAFE}_{\mathsf{WApp}}$ uses Jericho [22] to extract JavaScript code and its source location from an HTML source and passes them to the original JavaScript parser in SAFE. Then, it uses CyberNeko [23] to construct a DOM tree from the HTML source and passes it to the `DOMModeler` and `DOMBuilder` modules, which create abstract models of browser environments and DOM trees respectively and pass them to the analyzer as shown in Figure 3.

Then, we model the event system of browser environments to support interactions between web applications and users via events. While previous work [12] uses an event model where load event dispatches preceded dispatches of other events, we found that this model may produce unsound analysis results because dispatches of other events indeed can precede load events. Hence, we choose a more conservative event modeling for soundness: after analyzing a given program, it analyzes all event handlers registered by the program and safely combines their analysis results. Note that users may register event handlers not only statically using HTML attributes such as `onload` and `onclick` but also dynamically using the `addEventListener` API. $\mathsf{SAFE}_{\mathsf{WApp}}$ maintains a special abstract object, which serves as a table to keep all registered event handlers including the ones registered dynamically during analysis. The `EventCFGBuilder` module in Figure 3 adds an event execution loop at the end of the CFG of the program. The event loop represents program flows of all possible execution sequences of event handlers. Since combining analysis results may harm the analysis precision of the whole program, $\mathsf{SAFE}_{\mathsf{WApp}}$ provides a configurable option to enable or disable the event system.

C. DOM Prototype Objects and Browser APIs

To provide browser environments to the `Analyzer` module, the `DOMModeler` module adds DOM prototype objects and other built-in browser objects that have abstract browser APIs to the initial heap of the analyzer. Note that, unlike class-based languages like Java, JavaScript does not support classes and inheritance by subclassing but supports prototype-based object inheritance. When looking up a property in an object, if the object does not have the property, it looks up the property in the objects in its prototype chain.

The `DOMModeler` module creates abstract DOM prototype objects that DOM node objects inherit and built-in browser objects with browser APIs, constructs prototype relations among them, and puts them in the initial heap of the analyzer. We faithfully modeled, according to the DOM specifications from the W3C and WHATWG communities, about 130 abstract objects and 800 properties that include 196 among 200 properties that we found in more than 1,000 sites in the empirical study of Section IV-B. While the specifications also define some interfaces that instance node objects in a DOM tree should implement, they, however, do not clearly describe a hierarchy between DOM interfaces and objects implementing them. For example, a `<div>` tag in an HTML document creates an `HTMLDivElement` instance object, which should implement the `HTMLElement` interface; an object that implements the `HTMLElement` interface should implement the `Element` interface; likewise, an object that implements `Element` should implement the `Node` interface. However, the specifications do not describe how to implement such relations.

To provide the relationships between DOM objects in a precise and practical way, $\mathsf{SAFE}_{\mathsf{WApp}}$ models prototype relations between DOM interfaces following the implementation model of browsers rather than the specifications. In real implementation, while APIs defined in an interface are properties of the prototype object of the interface, fields defined in an interface may not be properties of the corresponding prototype object.

For instance, the `appendChild` API defined in the `Node` interface is a property of the `Node.prototype` object, but the `firstChild` field defined in the same `Node` interface becomes a property of all the instance node objects in a DOM tree.

D. DOM Tree Construction

The `DOMBuilder` module constructs an abstract DOM tree and adds it in the initial heap of the analyzer. $\mathsf{SAFE}_{\mathsf{WApp}}$ provides configurable DOM tree abstraction levels in three ways, two from the literature and one new:

- *Model A* keeps one single abstract node that approximates all kinds of DOM nodes regardless of HTML source structures like the model that GATEKEEPER [11] uses; all links to parent, children, and sibling nodes always point to the one single abstract node.
- *Model B* keeps one single abstract node for nodes in the same kind (for example, nodes with the same tag) but different abstract nodes for those in different kinds, regardless of HTML source structures like the model that TAJS [12] uses; all links to parent, children, and sibling nodes always point to all possible abstract nodes.
- *Model C* keeps a single abstract node for each DOM node by cloning the concrete DOM tree from HTML parsers; links to parent, children, and sibling nodes point to precise single abstract nodes initially, but they may be over-approximated during analysis.

While Model C provides the most precise DOM tree model, B the second, and A the least, Model A and B always keep the same numbers of abstract objects for a DOM tree regardless of an HTML source structure, which are usually smaller than the number of abstract objects in Model C; the number of abstract objects for a DOM tree varies depending on HTML source structures in Model C. Thus, three models can have tradeoffs between analysis precision and performance. Smaller numbers of abstract objects make an analysis domain smaller, which helps the analysis to reach a fixpoint earlier. Moreover, Model A and B make the implementation of abstract APIs for DOM tree search and manipulation simpler than Model C. While Model C implements actual search and manipulation on abstract DOM trees for the APIs, Model A and B simply return all possible abstract nodes for search and do nothing for manipulation, which improves analysis performance. We show the effect of each model on analysis precision and performance using analysis results of real-world applications in Section VII.

VI. APPLICATION OF $\mathsf{SAFE}_{\mathsf{WAPP}}$

Now that $\mathsf{SAFE}_{\mathsf{WApp}}$ provides an extensive model of browser environments, it extends the applicability of client applications built on top of SAFE from stand-alone JavaScript programs to web applications. Various client applications can utilize analysis results of $\mathsf{SAFE}_{\mathsf{WApp}}$ for program understanding, debugging, and optimizations. In this paper, we present a bug detector that detects JavaScript language-level bugs as an application of $\mathsf{SAFE}_{\mathsf{WApp}}$. The bug detector can detect the following 7 bugs: 4 errors and 3 warnings:

- AbsentVar signals undefined variable accesses, which cause the `ReferenceError` exception in JavaScript.
- CallNonFun signals function calls with non-function values, which cause the `TypeError` exception.
- NullOrUndef signals property accesses of `null` or `undefined`, which cause the `TypeError` exception.
- BinaryType signals non-object values used in the right-hand sides of the `in` or `instanceof` operator, which cause the `TypeError` exception.
- CondBranch signals conditional expressions with constant values or applications of the === operator that evaluate to always true or always false in `if` statements; subtleties in correctly using == and === operators can cause unintended behaviors as we showed in Section II.
- ConvertUndefToNum and PrimitiveToObj signal suspicious implicit type conversions from `undefined` to

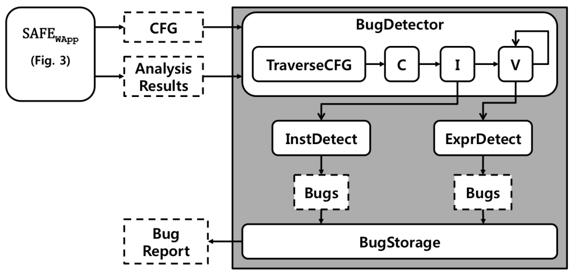


Fig. 4: Bug detector using analysis results form SAFE$_{\mathsf{WApp}}$

a number and from a primitive value to an object, respectively.

Figure 4 illustrates the overall structure of the bug detector; the shaded box indicates the bug detector with its main modules. It takes the CFG of a target program and its analysis results from SAFE$_{\mathsf{WApp}}$ as inputs and produces a bug report as an output. As with CFGs in traditional compilers and analyzers, a CFG node in SAFE$_{\mathsf{WApp}}$ corresponds to a basic block containing a list of instructions, and an instruction contains expressions that have no side effects on heaps (and abstract heaps) during evaluation (and analysis). Recall that the SAFE$_{\mathsf{WApp}}$ analyzer computes a summary map $\hat{s}$ from each program point (a node and a context) to an abstract heap at the program point as an analysis result. In order to perform bug detection efficiently, the bug detector first trims unnecessary information from the summary map $\hat{s}$. The BugDetector module refines $\hat{s}$ to another one $\hat{s}_b$ that maps a pair of an instruction i or an expression e and a context c to an input abstract heap $\hat{h}$ at the instruction i or expression e while traversing the CFG by the TraverseCFG module. Then, using $\hat{s}_b$, the C module inspects all CFG nodes and calls the I module when it inspects instructions in the nodes. Likewise, the I module inspects instructions and calls the V module when it inspects expressions in the instructions. While inspecting instructions and expressions, the I module and the V module call the InstDetect module and the ExprDetect module, respectively, to perform actual bug detection and pass detected bug information to the BugStorage module. Finally, the BugStorage module collects all the bugs detected and reports them in a user-readable format with source locations.

Because false positives are one of the main reasons why developers do not use static analysis tools [24], our bug detector reports only definite bugs by default and provides an additional developer mode to report all detected bugs that may include more false positives. Thus, with a default mode, the bug detector uses the following 3 restrictions on bug detection to reduce false alarms:

1) it does not consider CFG nodes in branches with uncertain conditions;
2) it reports only such bugs that are detected in all contexts; for example, if it detects a bug at the same program point in one call-context but not in another call-context, it does not report the bug;
3) it reports bugs only with definite value information; for example, if an abstract value $\hat{v}$ may be `null` or an object, then the bug detector does not report the NullOrUndef bug with the property access of $\hat{v}$.

Note that since SAFE$_{\mathsf{WApp}}$ soundly over-approximates program behaviors, bug detection with a developer mode will report more bugs possibly with more false positives.

Due to the space limitations, we present only the detection rules for NullOrUndef and BinaryType with a default mode in this paper, and we refer interested readers for details to the SAFE$_{\mathsf{WApp}}$ repository [16]. When evaluating a property access $e.p$, JavaScript first evaluates the expression e to a value and implicitly converts the value to an object. SAFE$_{\mathsf{WApp}}$ models such an implicit type conversion from a value to an object with the instruction $x := \widehat{\mathsf{toObject}}(e)$, and the InstDetect module of the bug detector inspects the instruction to detect any NullOrUndef errors as follows:

$$\begin{array}{l}\mathsf{InstDetect}[\![x := \widehat{\mathsf{toObject}}(e)]\!](\hat{s}_b) = \mathsf{signalBug}(\mathsf{NullOrUndef})\\ \quad \text{if } \forall c \in \mathsf{Context} : \exists\, \hat{h} \in \widehat{\mathsf{Heap}} \text{ and } \hat{v} \in \widehat{\mathsf{V}} \text{ such that}\\ \qquad \hat{s}_b(x := \widehat{\mathsf{toObject}}(e), c) = \hat{h} \ \wedge\ \hat{F}_e(e)(\hat{h}, c) = \hat{v} \ \wedge\\ \qquad (\hat{v}.\_1 = \top_{\text{Undef}} \ \vee\ \hat{v}.\_2 = \top_{\text{Null}}) \ \wedge\ \hat{v}.\_3 = \bot_{\text{Bool}} \ \wedge\\ \qquad \hat{v}.\_4 = \bot_{\text{Number}} \ \wedge\ \hat{v}.\_5 = \bot_{\text{String}} \ \wedge\ \hat{v}.\_6 = \varnothing\end{array}$$

where $\hat{F}_e(e)(\hat{h}, c)$ evaluates the expression e with the abstract heap $\hat{h}$ and the context c to an abstract value, and $\hat{v}.\_n$ accesses the n-th element in the tuple of the abstract values $\hat{v}$. The rule signals the NullOrUndef bug only when the expression e evaluates to either `undefined` or `null`, not to other values, in all contexts to minimize false positives.

To detect non-object values used in the right-hand sides of the `in` or `instanceof` operator, the ExprDetect module of the bug detector inspects binary operation applications as follows:

$$\begin{array}{l}\mathsf{ExprDetect}[\![e_1 \otimes e_2]\!](\hat{s}_b) = \mathsf{signalBug}(\mathsf{BinaryType})\\ \quad \text{if } \otimes \in \{\mathtt{in}, \mathtt{instanceof}\} \ \wedge\\ \qquad \forall c \in \mathsf{Context} : \exists\, \hat{h} \in \widehat{\mathsf{Heap}} \text{ and } \hat{v} \in \widehat{\mathsf{V}} \text{ such that}\\ \qquad \hat{s}_b(e_1 \otimes e_2, c) = \hat{h} \ \wedge\ \hat{F}_e(e_2)(\hat{h}, c) = \hat{v} \ \wedge\ \hat{v}.\_6 = \varnothing.\end{array}$$

The rule signals the BinaryType bug only when the right-hand side expression e_2 evaluates to a non-object value ($\hat{v}.\_6 = \varnothing$) in all contexts to minimize reporting of false positives. In the next section, we present bug detection results of our bug detector on 10 real-world applications.

VII. Experimental Evaluation

We evaluate SAFE$_{\mathsf{WApp}}$ with the analysis results of popular JavaScript libraries and websites. Our implementation of SAFE$_{\mathsf{WApp}}$ and target programs are publicly available [14]. Table IV summarizes the evaluation results. The first and second columns of the table show the 15 target programs of the evaluation and line numbers of their pretty-printed JavaScript sources, respectively. The first 5 targets are simple programs that just load the 5 most popular JavaScript libraries according to W3Techs, and the remaining 10 targets are the main web pages of the 10 most popular websites according to Alexa that we collected using source saving functions of browsers. We used JavaScript libraries because they have been the major targets for JavaScript analysis [5], [25], [26], and we selected web sites because JavaScript code in web pages may be the closest one to JavaScript web applications in the wild.

TABLE IV: Analysis results of the 5 most popular JavaScript libraries and main web pages of the 10 most popular websites with 3 different DOM tree abstraction models. Model A and B are previously used by GATEKEEPER [11] and TAJS [28], respectively, and Model C is proposed in this paper. The analysis time is in seconds with the timeout of 3 hours denoted by ✗. The analysis precision is measured with MD (multiple dereferences), MC (multiple calls), and PR (non-constant property accesses).

Target	LOC	Model A				Model B				Model C			
		Time	MD	MC	PR	Time	MD	MC	PR	Time	MD	MC	PR
jQuery 2.1.1	7,530	✗	416/3,059	7/392	60/177	✗	440/3,055	7/392	60/177	13.37	11/2,490	3/323	18/112
Modernizr 2.8.3	770	5.29	39/822	3/143	1/24	6.02	55/821	3/143	1/24	5.19	4/733	3/129	0/23
BootStrap 3.3.0	10,027	✗	649/5,101	16/662	73/325	✗	647/5,081	16/662	69/306	✗	561/4,985	19/622	90/294
Mootools 1.5.1	6,756	✗	64/305	5/38	6/11	✗	92/533	13/65	8/28	359.05	33/3,268	6/389	9/147
Prototype 1.7.2	7,517	42.59	60/2,609	11/342	11/64	40.82	55/2,556	8/333	8/61	36.27	20/2,469	5/326	7/57
`google.com`	3,698	✗	289/1,440	2/167	4/48	✗	290/1,440	2/167	4/48	2,662.22	274/5,916	10/748	35/147
`facebook.com`	7,510	✗	71/969	2/131	4/41	✗	71/969	2/131	4/41	✗	32/6,402	3/681	2/199
`youtube.com`	8,306	✗	253/4,764	3/621	38/176	✗	321/5,014	3/637	38/176	✗	153/5,591	4/764	30/195
`baidu.com`	6,117	✗	396/4,666	23/626	109/364	✗	488/4,680	23/629	109/364	✗	292/4,198	39/554	88/330
`yahoo.com`	16,852	✗	230/3,508	4/417	33/537	✗	241/3,532	4/320	33/537	7,298.93	153/8,876	5/700	17/638
`wikipedia.org`	382	6.90	19/495	6/98	3/24	8.62	44/495	5/98	3/24	127.01	36/458	5/91	2/22
`amazon.com`	7,076	✗	833/4,747	16/669	32/114	✗	862/4,744	16/669	32/114	✗	144/4,403	14/638	23/108
`twitter.com`	15,982	✗	291/6,143	11/664	47/214	✗	408/6,314	16/697	77/243	✗	418/8,402	50/987	84/411
`taobao.com`	6,300	✗	766/6,352	33/869	80/216	✗	811/6,367	33/870	80/216	✗	105/5,388	8/781	40/185
`qq.com`	17,706	✗	267/6,643	8/760	61/334	✗	315/6,637	8/760	61/334	✗	39/6,591	7/769	35/255
Average	–	–	309/3,441	10/439	37/177	–	342/3,482	10/438	39/181	–	151/4,678	12/566	32/208

Statically analyzing JavaScript code in web pages is full of challenges. One big challenge, also common in all static analyses, is dealing with dynamic code generation. JavaScript code in some websites is statically invisible since it is dynamically generated by calling functions such as `eval` and `Function` or loaded lazily by the `<script>` tag with JavaScript source code using `document.write` or `document.createElement` at run time. Because such code fragments are not available at compile time but appear at run time, they may invalidate static analysis results. Among the 10 most popular websites, We detected 4 `eval` calls in `youtube.com`, `baidu.com`, and `qq.com` during analysis. Since all calls are with constant string arguments, we could easily replace the `eval` function calls with other language constructs [27].

With the 15 target programs, we first evaluate how DOM tree abstraction levels affect analysis results with respect to performance and precision. We compare the analysis results of three different DOM tree models that SAFE$_{\text{WApp}}$ provides, Model A, B, and C described in Section V-D. Note that since Model A and B are DOM tree abstraction models previously used by GATEKEEPER [11] and TAJS [28], respectively, the evaluation results also allow us to compare existing models in the literature (Model A and B) and a new one (Model C). Next, we discuss the modeling coverage of all detected browser APIs during analysis, and we present bug detection results on the main web pages of the 10 most popular websites using our bug detector. For fair comparison, we used the same configurations with 10 call-context sensitivity and loop-sensitivity, but different ones only for DOM tree models. We conducted all experiments on a Linux x64 machine with 3.4GHz Intel Core i7 CPU and 32GB memory.

A. *DOM Tree Abstraction*

Table IV shows analysis results of the 15 target programs by the analysis of SAFE$_{\text{WApp}}$ with the timeout of 3 hours when we use three different DOM tree abstraction models. The first columns in the sections of Table IV for Model A, B, and C show the time in seconds that SAFE$_{\text{WApp}}$ took to analyze target programs. When an analysis does not complete within the timeout, we denote its time as ✗. The rest columns show three metrics that we used to measure the analysis precision:

- MD: the number of object dereferences with multiple object values (*multiple dereferences*) / the total number of analyzed object dereferences
- MC: the number of calls with multiple function values (*multiple calls*) / the total number of analyzed calls
- PR: the number of object property accesses with a property name approximated as non-constant (*non-constant property accesses*) / the total number of analyzed object property accesses except for those with constant property names such as `o["name"]`

The experimental results show that while Model A and B do not show significant differences in analysis performance and precision, Model C outperforms Model A and B in both performance and precision in most cases.

For the analysis performance, while using Model A and B can finish analysis of only 3 targets–Modernizr, Prototype, and `wikipedia.org`, using Model C can finish analysis of 4 more targets–jQuery, Mootools, `google.com`, and `yahoo.com`. For example, while analysis with Model A and B cannot analyze jQuery, the most popular JavaScript library with the market share of more than 90%, within the timeout of 3 hours, analysis with Model C finishes analyzing jQuery in 13.37 seconds. Note that `wikipedia.org` is an outlier in analysis results. Even though analysis with Model A and B can finish analyzing `wikipedia.org` in less than 9 seconds, it took more than 2 minutes with Model C. Because Model C provides precise DOM modeling, analysis using Model C involves overhead in maintaining precise DOM models, which may be an overkill for analyzing simple programs like `wikipedia.org`.

As for the analysis precision, all three metrics MD, MC, and PR on average (the last row of Table IV) show that Model C has the least ratios of multiple dereference, multiple call, and non-constant property accesses over corresponding total program points, although it has the most numbers of total program points in the ratios; for example, Model C has the least ratio 2.12% (12/566) on average in MC compared to 2.29% (10/436) and 2.28% (10/438) in Model A and B, but it has the most number of total program points as 566 compared to 436 and 438 in Model A and B, respectively. Note that the least ratios mean the most precise analysis results and the most numbers of total program points mean the best analysis coverage of target program points. We found that Model C analyzes more program points than Model B and C with the same timeout in most cases, which implies that Model C shows better analysis scalability in such cases; in case of `google.com`, for example, we found that analyses with Model A and B cover 10.79% and 10.83% of CFG nodes, respectively, while that with Model C covers 39.01% within the same timeout of 3 hours.

The results in Table IV indicate that the precise DOM modeling in Model C can improve the analysis performance in some JavaScript web applications. Unlike analysis of C-like languages where precise analyses often sacrifice the analysis performance, researchers [21], [5], [29] recently reported that high precision in static analysis of JavaScript applications can increase the analysis scalability significantly. For instance, low precision in analysis of higher-order function calls mixed with dynamic property accesses as in "`o[x](...)`" may lead to many spurious function calls increasing analysis computation immensely. Likewise, imprecise property values in DOM trees of Model A and B may harm the analysis scalability.

Finally, we investigated why $\mathsf{SAFE}_{\mathsf{WApp}}$ with configurable DOM tree models fails to complete the analysis of 8 target programs within the timeout. One reason was prevalent uses of statically indeterminate values such as `Math.random()` and browser environmental information. We observed that sound abstraction of such values as all possible ones would lead to state explosion during analysis. Another reason was the sound event system of $\mathsf{SAFE}_{\mathsf{WApp}}$, which considers all possible execution sequences of event handlers where the worst fixpoint computation time of the analysis exponentially increases as the number of registered event handlers increases. We found that `facebook.com` registers 29 mouse event handlers only with html attributes statically. We expect that user inputs for statically indeterminate values and more elaborate event systems would alleviate the scalability problem.

B. Modeling Coverage

To evaluate the effects of the proposed modeling mechanism, we measured the coverage of models encountered during analysis. We first identified "definite browser API calls" each of which has a single function value during analysis of 15 target programs using Model C within the timeout of 3 hours. Then, we checked whether API function calls encountered during analysis are covered in a hypothetical model which includes only the modeling of 52 APIs found in more than 1,000 websites from the empirical study of Section IV. We found that the model covers all APIs detected in 4 targets; the worst case was `youtube.com` where the model covered 18 APIs out of 24 encountered APIs (75.5%) and on average the model covers 90.33% of all APIs encountered in each target program. The experimental results indicate that modeling browser APIs based on empirical data is a promising approach.

C. Bug Detection

We present bug detection results in the main web pages of the 10 most popular websites using the bug detector we have built as an application of $\mathsf{SAFE}_{\mathsf{WApp}}$ (Section VI). The bug detector uses analysis results from $\mathsf{SAFE}_{\mathsf{WApp}}$ with the most precise DOM tree abstraction (Model C) with the timeout of 3 hours. It did not report any errors or warnings in `google.com` and `baidu.com` but it reported 3 errors and 23 warnings in the remaining 8 websites: 3 NullOrUndef, 19 CondBranch, 2 ConvertUndefToNum, and 2 PrimitiveToObj.

In addition to the bug in `wkipedia.org` we described in Section II, we found another interesting bug in `amazon.com` detected as a ConvertUndefToNum warning as follows:

```
var isTouchDevice =
  ... || navigator.MaxTouchPoints > 0 || ...
```

where the code checks if a current device is a touch device by checking if the maximum number of touch points in a current device is greater than 0. However, since browsers provide the maximum number of touch points with `navigator.maxTouchPoints` rather than `navigator.MaxTouchPoints`, the code contains a typing error with the capital `M`. Consequently, this bug causes the condition expression to always evaluate to false via the following evaluation steps: first, `navigator.MaxTouchPoints` evaluates to `undefined`, then the `undefined` value implicitly converts to a number, `NaN` in this case, for the comparison with 0, and `NaN > 0` always evaluates to false. The bug detector finds such suspicious implicit conversion from `undefined` to a number and signals the ConvertUndefToNum bug. After we reported the bug, Amazon is investigating it.

The bug detector finds suspicious conditional expressions that always evaluate to the same value in conditional branches, and it signals CondBranch warnings for such cases. For example, `if(true) {...}` from `facebook.com` is such a case. In this case, we can safely rewrite the `if` statement to its true-branch statement preserving the semantics of the program. Note that detecting and rewriting ever-true or ever-false conditional branches may enable web page optimization by reducing code size.

PrimitiveToObj warnings signal suspicious implicit type conversions from primitive values to objects. For example, `(+(new Date)).toString().slice(-4)` from `twitter.com` illustrates such a case. After creating an instance `Date` object, the code implicitly converts the object to a positive number by applying the `+` operator, which subsequently converts to a `Number` object for the `toString` method call. While the implicit conversion from a primitive number to an object in this code may be intentional, since such conversions often lead to unexpected behaviors, the bug detector reports them as the PrimitiveToObj warning.

Finally, we manually inspected all reported bug messages to check if they are true alarms. Using the developer mode in various browsers, we compared the bug detection results and real execution results to identify false alarms, which have inconsistent messages with the real execution results. We found that 6 bug reports out of 26 messages were false alarms. The false alarms mainly come from lazily loaded script code and incomplete analysis results due to timeout. Consider `Y.ModulePlatform.init(...)` from `yahoo.com` where `Y.ModulePlatform` is defined in a script loaded lazily from a different server. Since SAFE$_{\text{WApp}}$ does not have analysis results of such lazily loaded code, the analyzer considers its value as `undefined`, which makes the bug detector identify `Y.ModulePlatform.init` as an illegal property access of `undefined` reporting NullOrUndef. We can eliminate such false alarms if users provide lazily loaded scripts to SAFE$_{\text{WApp}}$ before analyzing the code. Another false alarm is `this.options.bufferSize<0` from `twitter.com`. Because the SAFE$_{\text{WApp}}$ considers `this.options.bufferSize` as `undefined`, the bug detector regards the code as a suspicious implicit type conversion from `undefined` to a number, and it signals the ConvertUndefToNum warning. However, we found that the property is actually defined in an event handler, which was not analyzed due to the timeout of 3 hours; the bug detector reported the false alarm since the SAFE$_{\text{WApp}}$ analyzer missed the event function during analysis,. We can eliminate such false alarms with longer timeout for analysis.

D. Threats to Validity

Soundness and precision. Although we faithfully modeled browser environments with frequently used APIs, web applications may use APIs not included in our modeling. In such cases, because SAFE$_{\text{WApp}}$ simply reports warning messages for the API uses, the analysis results of web applications using the APIs become unsound or imprecise. Systematic and reusable modeling techniques may reduce the modeling effort and enhance the soundness and precision of analysis results. Also, incomplete analysis results with timeouts may produce unsound results. More sophisticated modeling of the event system considering possible orders among events could improve the scalability and precision of the analysis as well as the soundness by completing the analysis within timeouts.

Bug detection. While SAFE$_{\text{WApp}}$ could detect some previously undiscovered bugs with warning messages that may lead to unintended behaviors, it did not find any true errors. However, because the target websites have been deployed and used by many people for long periods, it is highly likely that all critical errors like run-time exceptions may have been already reported and fixed. We believe that the SAFE$_{\text{WApp}}$ bug detector would be more useful to detect bugs in JavaScript web applications under development rather than after deployment. As SAFE$_{\text{WAPI}}$ [2], an extension of SAFE, analyzes API specifications additionally to detect incorrect uses of APIs in JavaScript web applications, we believe that integrating SAFE$_{\text{WAPI}}$ and SAFE$_{\text{WApp}}$ will enlarge a set of bugs that it can detect.

VIII. Related Work

Gatekeeper [11] combines a static points-to analysis of JavaScript programs in the web and run-time checks for unresolved names during static analysis. It models a DOM tree and browser APIs as a single node object and as mock-up objects, respectively. On the contrary, SAFE$_{\text{WApp}}$ is fully static, and it supports the full JavaScript with the faithful model of browser APIs frequently used in real websites. Note that SAFE$_{\text{WApp}}$ incorporates the DOM tree model of Gatekeeper as a configurable option.

Lerner *et al.* [9] formally modeled the event system of web browsers faithfully as specified in W3C DOM Level 3 Events. The model includes the complex event dispatch mechanism with event capturing and event bubbling. However, their model supports restricted subsets of DOM objects and the JavaScript language only with event-related features. Note that our event model does not need to consider the event capturing and bubbling mechanism since it abstracts all possible event dispatch and event handler execution.

Jensen *et al.* extended TAJS with a DOM and browser APIs modeling [12]. While their event modeling inspired ours, there are significant differences between our approach and theirs. First, they modeled browser environments based on selected specifications, but our modeling is based on extensive empirical data. Second, their modeling supports only simple DOM tree abstraction. TAJS abstracts HTML elements with the same tags as one single abstract object having a simple abstract domain for a DOM tree by sacrificing its tree structure. Because of this simple abstraction, all parent, children, and sibling links of a node in a DOM tree point to all possible node objects with imprecise information. Consequently, TAJS fails to give precise analysis results on simple programs that explore and manipulate DOM trees [3]. On the contrary, SAFE$_{\text{WApp}}$ provides configurable DOM tree abstraction, which supports not only the TAJS model but also a new DOM tree abstraction that captures the concrete DOM tree precisely at the initial state of the analyzer. In Section VII, we showed that high precision in DOM tree abstraction can significantly improve the analysis scalability for real-world JavaScript web applications.

IX. Conclusions

We presented, SAFE$_{\text{WApp}}$, an open-source static analysis framework for JavaScript web applications. It supports the full JavaScript language and features an extensive model of browser environments. Unlike previous work based on a few specifications, our DOM model is based on empirical data collected from the 9,465 most popular websites. It provides configurable DOM tree abstraction models in three levels, two existing models and one new model that captures the precise tree structure of a concrete DOM tree at the initial state of analysis. We evaluated how different DOM tree abstraction models affect the analysis performance and precision using real-world JavaScript web applications in the wild. The experimental results showed that precise DOM tree abstraction can significantly improve the analysis scalability. On top of SAFE$_{\text{WApp}}$, we have built a bug detector capable of detecting JavaScript language-level bugs as an application of SAFE$_{\text{WApp}}$. While investigating the main web pages of the 10 most popular websites using the bug detector, we found previously undiscovered bugs in `wikipedia.org` and `amazon.com`. Since our bug detector reports only definite bug messages by default, the number of bug messages was manageable as 26 in 10 target programs, and 76.9% of all bug messages were true alarms. We make the implementation of SAFE$_{\text{WApp}}$ and the bug detector open to the public.

Acknowledgment

This work is supported in part by National Research Foundation of Korea (Grant NRF-2014R1A2A2A01003235), Samsung Electronics, and Google Faculty Research Award.

References

[1] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[2] S. Bae, H. Cho, I. Lim, and S. Ryu, "$\mathsf{SAFE}_{\mathsf{WAPI}}$: Web API misuse detector for web applications," in *ESEC/FSE '14: Proceedings of the 22nd ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 2014.

[3] E. Andreasen and A. Møller, "Determinacy in static analysis for jQuery," in *OOPSLA'14: Proceedings of the 29th Annual Object-Oriented Programming Systems, Languages, and Applications*, 2014.

[4] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "JSAI: A static analysis platform for JavaScript," in *FSE '14: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[5] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, "Correlation tracking for points-to analysis of Javascript," in *ECOOP'12: Proceedings of the 26th European Conference on Object-Oriented Programming*, 2012.

[6] W3C, "Document Object Model Activity Statement," http://www.w3.org/DOM/Activity.

[7] WHATWG, "HTML Living Standard," http://www.whatwg.org/specs/web-apps/current-work/multipage/.

[8] P. A. Gardner, G. D. Smith, M. J. Wheelhouse, and U. D. Zarfaty, "Local hoare reasoning about DOM," in *PODS'08: Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM, 2008.

[9] B. S. Lerner, M. J. Carroll, D. P. Kimmel, H. Q.-D. La Vallee, and S. Krishnamurthi, "Modeling and reasoning about DOM events," in *WebApps'12: Proceedings of the 3rd USENIX Conference on Web Application Development*. USENIX Association, 2012.

[10] D. Yu, A. Chander, N. Islam, and I. Serikov, "JavaScript instrumentation for browser security," in *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2007.

[11] S. Guarnieri and B. Livshits, "GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code," in *SSYM '09: Proceedings of the 18th Conference on USENIX Security Symposium*. USENIX Association, 2009.

[12] S. H. Jensen, M. Madsen, and A. Møller, "Modeling the HTML DOM and browser API in static analysis of JavaScript web applications," in *ESEC/FSE '11: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 2011.

[13] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript," in *FOOL'12: International Workshop on Foundations of Object Oriented Languages*, 2012.

[14] KAIST PLRG, "Research material," http://plrg.kaist.ac.kr/pch.

[15] M. Madsen, B. Livshits, and M. Fanning, "Practical static analysis of JavaScript applications in the presence of frameworks and libraries," in *ESEC/FSE 2013: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013.

[16] KAIST PLRG, "SAFE: JavaScript analysis framework," http://safe.kaist.ac.kr, 2013.

[17] S. Ryu, J. Choi, W. Choi, Y. Ko, H. Lee, and C. Park, "The SAFE specification," https://github.com/sukyoung/safe/blob/master/doc/manual/safe.pdf, 2015.

[18] ECMA, "ECMA-262: ECMAScript Language Specification. Edition 5.1," 2011.

[19] P. Heidegger and P. Thiemann, "Recency types for analyzing scripting languages," in *ECOOP '10: Proceedings of the 24th European Conference on Object-Oriented Programming*, 2010.

[20] R. Mangal, M. Naik, and H. Yang, "A correspondence between two approaches to interprocedural analysis in the presence of join," in *ESOP 2014: Proceedings of the 23rd European Symposium on Programming*. Springer, 2014.

[21] C. Park and S. Ryu, "Scalable and precise static analysis of JavaScript applications via loop-sensitivity," in *ECOOP'15: Proceedings of the European Conference on Object-Oriented Programming*. Springer LNCS, 2015.

[22] "Jericho HTML parser," http://jericho.htmlparser.net.

[23] "Cyberneko HTML parser," http://nekohtml.sourceforge.net.

[24] B. Johnson, E. M.-H. Yoonki Song, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *ICSE'13: Proceedings of the International Conference on Software Engineering*, 2013.

[25] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Dynamic determinacy analysis," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[26] A. Feldthaus and A. Møller, "Checking correctness of TypeScript interfaces for JavaScript libraries," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2014.

[27] S. H. Jensen, P. A. Jonsson, and A. Møller, "Remedying the eval that men do," in *ISSTA 2012: Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012.

[28] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *SAS '09: Proceedings of the 16th International Symposium on Static Analysis*, 2009.

[29] E. Andreasen and A. Møller, "Determinacy in static analysis for jQuery," in *OOPSLA'14: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2014.