2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion

Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild

Joonyoung Park
KAIST
gmb55@kaist.ac.kr

Inho Lim
Samsung Electronics
inho0212.lim@samsung.com

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

ABSTRACT

Now that HTML5 technologies are everywhere from web services to various platforms, assuring quality of web applications becomes very important. While web application developers use syntactic checkers and type-related bug detectors, extremely dynamic features and diverse execution environments of web applications make it particularly difficult to statically analyze them leading to too many false positives. Recently, researchers have developed static analyzers for JavaScript web applications addressing quirky JavaScript language semantics and browser environments, but they lack empirical studies on the practicality of such analyzers.

In this paper, we collect 30 JavaScript web applications in the wild, analyze them using SAFE, the state-of-the-art JavaScript static analyzer with bug detection, and investigate false positives in the analysis results. After manually inspecting them, we classify 7 reasons that cause the false positives: W3C APIs, browser-specific APIs, JavaScript library APIs, dynamic file loading, dynamic code generation, asynchronous calls, and others. Among them, we identify 4 cases which are the sources of false positives that we can practically reduce. Rather than striving for sound analysis with unrealistic assumptions, we choose to be intentionally unsound to analyze web applications in the real world with less false positives. Our evaluation shows that the approach effectively reduces false positives in statically analyzing web applications in the wild.

CCS Concepts

•**Software and its engineering** → *Software testing and debugging;*

Keywords

Static analysis; JavaScript; web applications; false positives

1. INTRODUCTION

ICSE '16 Companion, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2889160.2889227

Using multiple devices with multiple platforms becomes normal, which makes cross-platform web applications prevalent on a variety of platforms including the web and mobile devices. Web applications are built on top of HTML5 [2] technologies containing HTML, CSS, and JavaScript; they can run anywhere that supports browsers. Because they are cross-platform, end-users can enjoy the same applications on different devices, and developers can build one application for multiple platforms. Indeed, major operating systems support most browsers and various mobile platforms such as Tizen[1], Chromium OS[2], and Firefox OS[3].

The more web applications become prevalent, the more quality assurance of web applications gets important, but developer tools for building reliable web applications are not yet ready. While web application developers use syntactic checkers like JSLint[4], extremely dynamic and functional features of JavaScript make it difficult to syntactically analyze them precisely. Researchers have reported that the lack of static types, dynamic addition and removal of object properties, object property names as first-class values, and dynamic code generation by `eval`-like functions all contribute to the complication of JavaScript static analysis [27]. Analysis of such features are beyond the scope of syntactic checking. In addition, diverse execution environments of web applications make the situation even worse because different environments provide different sets of functionalities that are targets of analysis [6].

Recent development in static analysis of JavaScript web applications enables analyzers to address quirky JavaScript language semantics and different browser environments, but their usability has not been evaluated with web applications in the wild. WALA [10, 29], the first static analyzer for JavaScript applications, provides a flow-insensitive points-to analysis, but it does not support the HTML DOM and the browser APIs required for analyzing web applications. TAJS [23, 13, 12] is based on the abstract interpretation framework [8], and it supports a flow-sensitive type analysis with a limited model of web documents. SAFE [15, 18] is similar to TAJS but it additionally supports detection of Web API misuses [6] and a systematic modeling of web documents [25]. They are actively expanding the set of JavaScript web applications that they can analyze, but they are still in the stage of striving for producing *sound* but possibly imprecise analysis results with unrealistic assumptions.

[1] https://www.tizen.org
[2] https://www.chromium.org/chromium-os
[3] https://www.mozilla.org/en-US/firefox/os
[4] http://www.jslint.com

```
1  // Tizen SDK Sample Application : ClockWidget
2  window.requestAnimationFrame =
3    window.requestAnimationFrame ||
4    window.webkitRequestAnimationFrame || // Chrome,
         Opera
5    window.mozRequestAnimationFrame ||    // FireFox
6    window.oRequestAnimationFrame ||      // Old
         Opera
7    window.msRequestAnimationFrame ||     // IE
8    function (callback) {
9      use strict;
10     window.setTimeout(callback, 1000 / 60);
11   };
```

Figure 1: Browser-specific APIs in web applications

For instance, the model of web documents in TAJS is based on assumptions like "the execution order of event handlers is often not crucial for the analysis precision [12]," which is invalidated by recent studies [19, 4]. Because researchers found that software developers tend to avoid using static analyzers when they report too many false positives [14, 28], reducing false positives in reported alarms is one of the essential keys of *practically usable* tools.

In this paper, we collect and study 30 JavaScript web applications in the wild, analyze them using SAFE, and investigate false positives in the analysis results. We use SAFE for our study because it can statically analyze the most JavaScript web applications among existing analyzers [24, 25]. To understand the usability of the state-of-the-art JavaScript static analyzer, we manually investigated reported alarms from SAFE, inspected false positives in the analysis results, and classified them into 7 cases in terms of the reasons that cause them: W3C APIs, browser-specific APIs, JavaScript library APIs, dynamic file loading, dynamic code generation, asynchronous function calls, and other reasons. We observed that we might be able to eliminate certain classes of false positives effectively while sacrificing the analysis soundness [20]. For example, utilizing dynamic information of various execution environments in static analysis would reduce false positives in analysis results.

For example, let us consider browser-specific APIs. Different browsers may use different names for API functions with similar functionalities, which introduces particular coding idioms in web applications. The code excerpt from a Tizen SDK sample application[5] in Figure 1 illustrates such a case. The global function `requestAnimationFrame` (line 2) is initialized to one of browser-specific API functions (lines 4–7) or a function using `setTimeout` (lines 8–11). Depending on browsers that execute the application and depending on specific versions of the same browser, property names like `webkitRequestAnimationFrame` and `oRequestAnimationFrame` may be present or absent. The idiom supports browser compatibility between different versions and different browsers. However, because most static analyzers consider all possible execution environments to analyze them soundly, they may report that `window.oRequestAnimationFrame` (line 6) accesses an absent property `oRequestAnimationFrame`, because `window` may not have the `oRequestAnimationFrame` property in browsers except for old Opera browsers.

A possible solution to this problem is to limit targets of static analysis to specific execution environments. By fixing target environments, static analyzers do not need to consider all possible APIs but browser-specific APIs and perform more precise bug detection reporting less false positives. Such analysis results may not be sound in the sense that they do not subsume all possible execution flows, but a set of analysis results for specific execution environments would collectively subsume possible execution flows for the environments. Thus, one may want to get specific execution environments by taking their snapshots dynamically [17, 16] and use them to specialize analysis targets. Such dynamic information may include dynamically loaded files that are not available statically. By analyzing previously missed files, analyzers may enjoy not only improved precision but also improved coverage of analysis targets.

Thus, among 7 reasons for false positives in static analysis of real-world web applications, we discuss 4 cases where we can reduce false positives resulting from them. To remove false positives due to W3C and browser-specific APIs, we propose a mechanism to take snapshots of execution environments dynamically and to use them in static analysis. For false positives due to JavaScript library APIs, we present a method to selectively simulate the functionalities by using their types. For dynamic file loading, we describe a way to record dynamically loaded files, and we show how to utilize such information to improve the precision of static analysis. Our evaluation with a prototype implementation shows that the proposed approach effectively reduces false positives in statically detecting bugs in web applications in the wild.

Our paper makes the following contributions:

- We collect and investigate 30 JavaScript web applications in the wild. We select Tizen web applications including market applications that can run on various devices like smart TVs[6], smart watches[7], and smart phones[8], and we analyze them using the state-of-the-art JavaScript static analyzer, SAFE.
- We manually inspect every alarm reported by SAFE for 30 target applications. We identify false positives and their root causes, and we classify them into 7 kinds.
- We propose techniques to reduce false positives in 4 cases taking advantage of dynamic information and type-based modeling.
- We evaluate the techniques with real-world web applications and show that they indeed improve analysis precision by reducing false positives with modest performance overhead.

The remainder of this paper is organized as follows. In Section 2, we provide background of our research by describing the definition and nature of web applications, the static bug detector for web applications, SAFE, and the false positive issues of SAFE. Section 3 explains web application characteristics that are challenging to static analysis using concrete examples. We propose our mechanism to reduce false positives in static analysis by using dynamic information and type-based modeling in Section 4, and we evaluate our mechanism using real-world web applications in Section 5. We discuss lessons learned in Section 6 and related work in Section 7, and we conclude in Section 8.

[5]https://developer.tizen.org/documentation/tutorials/web-application/w3chtml5supplementary-features/graphics/html5-canvas/task-clock-widget

[6]https://www.samsungdforum.com/Tizen/Spec

[7]http://www.tizenexperts.com/category/smart-watch/

[8]http://www.tizenphones.com

Table 1: Alarms detected by the SAFE bug detector

Alarm name	Error type	Description
AbsentReadVariable	ReferenceError	Reading an absent variable.
CallNonConstructor	TypeError	Calling a non-constructor object as a constructor.
CallNonFunction	TypeError	Calling a non-function object as a function.
ObjectNullOrUndef	TypeError	Accessing a property whose value is null or undefined.
AbsentReadProperty	Warning	Reading an absent property of an object.
BuiltinWrongArgType	Warning	Wrong types of arguments to built-in API functions.
ConvertUndefToNum	Warning	Converting `undefined` to a number.
ImplicitTypeConvert	Warning	Implicit type conversion in equality comparison.
GlobalThis	Warning	Referring the global object by `this`.

2. BACKGROUND

2.1 Definition and Nature of Web Applications

Web applications often consist of HTML documents for structuring contents, CSS for visual rendering, and JavaScript code for controlling user interaction. In this paper, we focus on web applications that satisfy the following conditions:

1. They use only HTML, CSS, and JavaScript. We do not consider applications that use JavaScript and other programming languages together.
2. They use only local resources including the main entry `index.html` file. We do not consider web applications that use resources over network like websites.

Among various nature of web applications, we focus on heavy use of *API and library functions* and *various kinds of dynamic data*. In addition to HTML/DOM and CSS APIs, web applications often use multiple JavaScript libraries to support complex functionalities including cross-platform capabilities. According to W^{3}Techs, about 68.2% of all the websites use any kind of JavaScript libraries, and about 65.0% of the websites use the jQuery[9] library, which is a JavaScript library market share of 95.3%[10]. To support functionalities beyond the scope of JavaScript, web applications use many browser-specific APIs written in native code. Also, web applications often load some files dynamically after loading their initial pages to reduce the overall loading time.

2.2 SAFE Bug Detector for JavaScript Web Applications

We choose SAFE for this study because it can analyze the most JavaScript web applications statically among existing analyzers [24, 25]. To help developers build reliable web applications, the SAFE static analyzer supports bug detection. Throughout this paper, we refer to the bug detection capability of SAFE *the SAFE bug detector*. The SAFE bug detector statically analyzes web applications and reports possible errors and warnings (collectively called alarms) as summarized in Table 1. While the SAFE bug detector can detect web API misuses as well [6], we focus on 9 alarms in this paper: one ReferenceError, three TypeErrors, and five Warnings. While ReferenceError and TypeError are defined in the language specification [1], which make JavaScript engines in browsers throw exceptions, Warnings are not defined in the specification but signaled by the SAFE bug detector as error-prone code patterns.

[9]http://jquery.com

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

Table 2: Alarms from 30 Tizen web applications

Application (LOC)	# Total	# TP	# FP	# MF
Annex (2571)	17	5	12	0
Bubblewrap (7222)	24	1	23	0
Calculator (9062)	0	0	0	0
CountingBeads (6949)	23	2	21	0
Flashcards(2302)	0	0	0	9
Go (2644)	1	0	1	0
HangOnMan (1326)	26	3	23	0
Mancala (1546)	37	3	34	0
MemoryGameOlderKids (6955)	42	15	27	0
MemoryMatch (3501)	0	0	0	12
Numeroo (8245)	19	2	17	0
Rabbit (1403)	24	6	18	0
ShoppingList (11034)	16	2	14	0
SliderPuzzle (11484)	0	0	0	0
SweetSpot (2356)	0	0	0	0
Tenframe (9238)	0	0	0	14
TodoList (18006)	42	0	42	0
BluetoothChat (1622)	5	1	4	12
CallLog (1258)	11	2	9	7
EventManager (1453)	6	1	5	6
AreaConverter (492)	11	0	11	0
ComicsM (13032)	2	0	2	0
FullHouse (439)	1	0	1	0
FusuiColorCompass (29150)	14	0	14	0
HandyPlanner (557)	5	0	5	0
IM+ (33196)	15	4	11	0
LoveCalculator (64)	15	0	15	0
MahjongBook (5830)	24	6	18	0
SimpleCounter (6000)	3	2	1	0
UBRadioactive (32377)	6	0	6	3
Total	389	55	334	63

The SAFE bug detector can analyze real-world web applications using various APIs including both native and non-native APIs via modeling. For native APIs that are implemented in native code, SAFE uses manually modeled information to analyze their functionality. In addition to standard HTML5 APIs, SAFE models some of browser-specific APIs as well [25]. While manually modeling such APIs is labor-intensive and error-prone, it is indispensable for precise analysis. Even for non-native library APIs that are implemented in JavaScript, SAFE models frequently used APIs like jQuery to focus on analyzing user code rather than library code. Depending on client analysis, SAFE provides options to select whether to analyze the jQuery library as well [24] or to use its model. Furthermore, SAFE also models the most frequently used JavaScript libraries from registered web applications in the Tizen store based on empirically collected data [6]. For more than 1800 APIs from about 35 libraries collected, SAFE recognizes them and replaces them with modeled information before analysis.

Table 3: Causes of false positives in static analysis

Causes	# FP	Subcategory	# FP
W3C APIs	160	imprecisely modeled	22
		missing	138
Browser-pecific APIs	19	imprecisely modeled	0
		missing	19
JavaScript library APIs	46	imprecisely modeled	44
		missing	2
Dynamic file loading	24		
Dynamic code generation	6		
Asynchronous calls	89		
Others	0		
Total	344		

2.3 False Positives from the SAFE Bug Detector

We use the SAFE bug detector as a baseline analyzer to evaluate our research results. Table 2 summarizes the bug detection results for 30 Tizen web applications with the timeout of 2000 seconds. We collect 17 Tizen sample cross-platform web applications, 3 Tizen-specific sample web applications[11], and 10 Tizen market web applications. While the source of 20 sample applications are publicly available, the source of 10 market web applications are not. To evaluate the analysis precision, we manually verified all the alarms from user code reported by the SAFE bug detector. Unfortunately, out of 389 alarms, only 55 are true alarms and the remaining 334 alarms are all false positives. Moreover, 63 files are not included in analysis of 7 web applications colored gray, which makes the analysis results unsound.

3. CASE STUDY: CAUSES OF FALSE POSITIVES

To understand *concrete and realistic* reasons that cause false positives in static analysis of web applications *in practice*, we investigated all 334 false alarms reported from 30 web applications. We classify them into 6 characteristics: 3 due to API modeling issues and 3 due to dynamic features. For the characteristics due to API modeling issues, we categorize them further into 2 subcategories: either due to imprecisely modeled APIs or rather due to missing APIs. We found that none of 334 false alarms are because of JavaScript language features. In this section, we discuss each characteristic using code excerpts from the target web applications.

3.1 W3C APIs

As we discussed so far, modeling W3C APIs completely and precisely is a key for static analysis of web applications. However, due to the labor-intensive nature of modeling and a large body of APIs, SAFE provides selective modeling of APIs based on empirical study of frequently used W3C APIs [25]. When APIs are not modeled in SAFE (missing APIs), their uses may be reported as accesses to absent properties leading to false positives. When SAFE does not provide precise API modeling (imprecisely modeled APIs), analysis results may be also imprecise resulting in false positives. As Table 3 denotes, issues with W3C API modeling are the major source of false alarms for our 30 subjects. Among 344 false positives, 160 are due to W3C APIs, among which 22 are due to imprecise modeling and the rest 138 are due to missing APIs. For example, Figure 2

[11]https://developer.tizen.org/downloads/sample-web-applications

```
1  // ShoppingList, main.js
2  ShoppingListApp = new function() {
3    var self = this;
4    // Init the application
5    // when the onLoad event is received.
6    self.initOnLoad = function() {
7      ...
8      self.buttonClick01Audio = new Audio(); // init
9      self.buttonClick01Audio.src =
10                     "./audio/ButtonClick_01.ogg";
11     self.buttonClick02Audio = new Audio();
12     self.buttonClick02Audio.src =
13                     "./audio/ButtonClick_01.ogg";
14     self.buttonClick03Audio = new Audio();
15     self.buttonClick03Audio.src =
16                     "./audio/ButtonClick_01.ogg";
17   }
18 }
```

Figure 2: The missing `Audio` API introduces false alarms.

```
1  // TodoList, helper.js
2  sortByTime: function (todos) {
3    var sortedArray = [], morningItems = [];
4    var afternoonItems = [], eveningItems = [];
5    $.each(todos, function() {
6     if(this.period == TodoItem.PeriodEnum.MORNING)
7       morningItems.push(this);
8     if(this.period == TodoItem.PeriodEnum.AFTERNOON)
9       afternoonItems.push(this);
10    if(this.period == TodoItem.PeriodEnum.EVENING)
11      eveningItems.push(this);
12   });
13 }
```

Figure 3: Imprecise modeling of a JavaScript library API produces false alarms.

shows that the ShoppingList application uses the `Audio` API defined in HTML5. However, because SAFE does not provide any modeling of `Audio`, construction of audio objects (lines 8, 11, and 14) returns `undefined`, which makes the values of all `self.buttonClick01Audio`, `buttonClick02Audio`, and `buttonClick03Audio` be `undefined`. Due to missing `Audio` API, SAFE reports false ObjectNullOrUndef alarms for any use of such objects. We found that 39 false positives in Table 3 are due to missing `Audio`.

3.2 Browser-Specific APIs

In addition to the standard W3C APIs, different browsers support their own APIs as well to provide advanced but not yet standardized features. To support multiple browsers with a single web application, developers often use programming idioms that first check browser-specific APIs and then implement different behaviors. Figure 1 in Section 1 is such an example. Similarly for other kinds of APIs, modeling browser-specific APIs requires much cost and efforts. Because SAFE does not support browser-specific APIs much, analysis of web applications that use browser-specific APIs may produce many false positives.

3.3 JavaScript Library APIs

Because analysis of complex JavaScript libraries like jQuery, Prototype, and MooTools does not produce precise analysis results efficiently, SAFE models widely used libraries for better analysis results. Like for W3C APIs and browser-specific APIs, modeling JavaScript libraries is costly and labor-intensive. Worse, most of them do not have API specifications unlike W3C APIs. Figure 3 illustrates that impre-

```
1  // CallLog
2  // index.html
3  <script src="./lib/.../js/jquery.min.js"></script>
4  <script src="./lib/.../js/tau.min.js"></script>
5  <script src="./js/main.js"></script>

6  // app.js
7  var App = null;
8  (function strict() {
9    'use strict';
10   App = function App() {
11     this.configData = {};
12   };
13   App.prototype = {
14     requires: [
15       'js/app.config.js',
16       'js/app.model.js',
17       'js/app.ui.js',
18       ...
19     ],
20     ...
21   }
22 }());

23 // main.js - Load dependencies
24 ({
25   init: function init() {
26     var self = this;
27     $.getScript('js/app.js')
28       .done(function onAppJsLoad() {
29       // once the app is loaded,
30       // create the app object
31       // and load the libraries
32         app = new App();
33         self.loadLibs();
34       })
35       .fail(this.onGetScriptError);
36   },
37
38   loadLibs: function loadLibs() {
39     var loadedLibs = 0;
40     if ($.isArray(app.requires)) {
41       aa.each(app.requires,
42         function onScriptIterate(index, filename) {
43          aa.getScript(filename)
44          .done(function onScriptLoad() {
45            loadedLibs += 1;
46            if (loadedLibs >= app.requires.length) {
47              // all dependencies are loaded,
48              app.init();
49            }
50          }).fail(this.onGetScriptError);
51         });
52     }
53   },
54   ...
55 }).init();
```

Figure 4: Dynamic file loading complicates static analysis.

cise modeling of JavaScript library APIs introduces many false positives. The TodoList application defines a function `sortByTime` that sorts DOM elements in order using the jQuery function `$.each` (line 5). According to the jQuery specification, the value of `this` inside `each` denotes each DOM element. However, because SAFE does not model the semantics precisely, each use of `this.period` (lines 6, 8, and 10) signals false GlobalThis and AbsentReadProperty.

3.4 Dynamic File Loading

As we discussed, web applications often load files dynamically to improve the application response time[12]. We observed that 7 out of 30 Tizen web applications load files dynamically. Because dynamically loaded files are not available statically and which files are loaded dynamically is not statically known, it is a very difficult challenge for static analysis. Thus, the SAFE bug detector analyzes only the embedded code and imported files in the main entry `index.html` file, and some frequently-used, specific patterns including `$.getScript` of jQuery with string literal arguments. For example, Figure 4 shows excerpts from several files of the CallLog application. The `index.html` file loads the `main.js` file (line 5). Then `main.js` calls `init` (line 55) defined in it (lines 25 – 36), which calls `$.getScript` (line 27) to load `app.js` dynamically. After loading `app.js` successfully, it calls `self.loadLibs` (line 33) defined in it (lines 38 – 53) via callback invocation, which loads files listed in `app.requires` (lines 41 – 51) in `app.js`. Finally, `app.js` enumerates several files to load (lines 15 – 18). Because of the complex dependency, the SAFE bug detector does not analyze the files loaded dynamically, and any accesses to variables defined in them result in false AbsentReadVariable alarms.

[12]https://developer.tizen.org/documentation/articles/performance-guide-tizen-web-applications2-rendering

3.5 Dynamic Code Generation

The extremely dynamic features of JavaScript allow developers to create code and data at run time. The notorious `eval` function takes a string argument that denotes a piece of JavaScript code, transforms the string value to a runnable JavaScript code, and evaluates the code. Other features like `Function` objects, JSON data, and the `setInterval` function incur similar problems. Researchers have proposed various approaches to analyze the `eval`-like functions statically [26, 11, 21], but they are inherently unsound missing JavaScript code to analyze statically. SAFE also addresses specific forms of dynamic code generation with string constants [11], but it apparently is not enough to reduce false positives in practice.

3.6 Asynchronous Function Calls

In addition to dynamic file loading and dynamic code generation, web applications use various patterns for asynchronous function calls.

3.6.1 *Event Function Calls*

Most web applications heavily interact with end-users. Each button on a screen has a registered event handler function, and call sequences between event handler functions should follow the logical orders between buttons designed by application developers. Depending on application states during evaluation, some buttons may be disabled and event handler functions registered on them may not be callable.

Because event handler function calls are very much dynamic interacting with user inputs, statically analyzing them precisely is extremely challenging. At the same time, imprecisely analyzing them easily blows execution flows to analyze, which harms both analysis precision and performance.

Consider the code from the HangOnMan game application in Figure 5. HangOnMan registers 22 event handler functions to various buttons and HTML elements. During evaluation, the event handler function registered to `"DOMContentLoaded"` (lines 4 – 11) gets evaluated first and initializes the `answer` object by calling the `initAnswer` function (line 8). Because the `answer` object stores the game state, many event handler functions access and modify `answer` during evaluation of the game. For example, the `giveUpConfirm` function

```
1  // HangOnMan, hangonman.js
2  var answer; // Game state
3  // Event Handler 1
4  window.addEventListener("DOMContentLoaded",
5                          function(event) {
6      ...
7      initDialogs();
8      answer = initAnswer(word);
9      ...
10     document.addEventListener("keyup",handleKeyUp,
           true);
11   }, false);
12 ...
13 function initAnswer (answerString) {
14   ...
15   var answer = {
16     "string": words.join(" "), "letters": "",
17     "numRight": 0, "data": []
18   };
19   ...
20   var letterElem = document.createElement("div");
21   var letter = words[i]
22   if (letter.match(alphabetRegex))
23     // 'answer' initialization
24     answer.letters += letter;
25     answer.data.push({ "elem": letterElem });
26   }
27   ...
28   return answer;
29 }
30 ...
31 // Event Handler 2
32 function giveUpConfirm (event){
33   ...
34   for (var i = 0; i < answer.letters.length; ++i) {
35    answer.data[i].elem.innerText=answer.letters[i];
36   }
37   ...
38 }
39 ...
40 // Event Handler 3
41 function chooseLetter (index) {
42   ...
43   if (answer.string.indexOf(letter) >= 0)
44   ...
45 }
```

Figure 5: Event function call sequences make static analysis results imprecise.

modifies `answer` (line 35) and the `chooseLetter` function accesses `answer` (line 43). However, because SAFE does not analyze the event function call sequence precisely, it does not analyze that `answer` is always initialized before calling `giveUpConfirm` or `chooseLetter`. Thus, it reports false alarms that `answer` may be undefined inside such event functions.

3.6.2 AJAX Programming

To take the best advantage of the Web, many web applications interact with Web Services to exchange multiple forms of data to provide complex user scenarios. Also, because JavaScript is not well equipped for manipulating big data, web applications store large data in separate files and use file I/O on demand to maintain such data efficiently. To support such functionality, AJAX programming like using `XMLHttpRequest` is frequently used.

For example, Figure 6 shows how the Numeroo application uses dynamically generated data received over a network. The `randomPuzzle` function takes `difficulty` as a user input (line 3) and requests an appropriate file from a local path (line 5) via `XMLHttpRequest` (lines 6 – 7). When the request completes, the event handler function registered on `onload` (lines 8 – 12) gets invoked, and it stores a random puzzle

```
1  // Numeroo, numeroo-engine.js
2  NumberPlaceGame.prototype.randomPuzzle =
3   function(difficulty) {
4    var self = this;
5    var file = "data/"+difficulty;    //file path
6    var request = new XMLHttpRequest();
7    request.open("GET", file, false);
8    request.onload = function(e) {
9     var requestStr = this.responseText.split("\n");
10    var l = (Math.random()*(requestStr.length-1))|0;
11    self.puzzle = requestStr[l].split(" ");
12   }
13   request.send();
14  }
15 ...
16 for(i = 0; i < 81; i++) {
17   var v = parseInt(this.puzzle[i]);
18     // generate a new grid
19     ...
20 }
```

Figure 6: Dynamically generated data via `XMLHttpRequest` complicates static analysis.

data in `self.puzzle` so that the game initialization code can use it to generate a game board (line 17). Because several data are received and generated dynamically, statically analyzing them is not plausible. Manually modeling such semantics is not trivial either because programming patterns to implement AJAX programming are numerous including `XMLHttpRequest` and `jQuery.ajax`.

4. REDUCING FALSE POSITIVES IN STATIC ANALYSIS

After investigating 7 challenging characteristics for static analysis explained in Section 3, we observed that we can reduce false positives resulting from 4 cases: W3C APIs, browser-specific APIs, JavaScript library APIs, and dynamic file loading. To improve analysis of W3C and browser-specific APIs, we use dynamic snapshot of execution environments. To enhance analysis of JavaScript library APIs, we model them using their type information. To analyze dynamically loaded files, we collect source code paths dynamically.

4.1 Snapshot of Execution Environments

Because the analysis precision of web applications heavily relies on available information about execution environments like specific browsers or platforms, we propose to utilize such dynamic information to improve the static analysis precision. By taking dynamic information of execution environments, static analysis may lose some information about other execution environments but it will produce more precise analysis results for given execution environments. Static analysis can use any existing modeling, and dynamic information can aid static analysis to fill missing W3C and browser-specific APIs.

To capture a specific execution environment of web applications, we "dump" all reachable information from the top-level `window` object, and we call the dumped dynamic information *snapshot*. Our implementation is written in JavaScript itself; starting from the `window` object, it gets a list of properties in the object via the `Object.getOwnPropertyNames` function, and dumps their information recursively traversing them. Note that because JavaScript is very flexible, our snapshot can include even JavaScript function bodies and their execution environments as well.

Table 4: Effects of using dynamic information and type-based modeling in static analysis of web applications

Application	Precision							Coverage			Scalability		
	Before		After		Change			Before		After	Before	After	
	# TP	# FP	# TP	# FP	# NTP (Good)	# RFP (Good)	# NFP (New)	# Add Files	# Anal. Funs	# Anal. Funs	Time (sec)	Time (sec)	Ratio
Annex	5	12	6	7	1	5	0	0	29	29	✗	✗	
Bubblewrap	1	23	1	4	0	19	0	0	42	42	1836.48	✗	
CountingBeads	2	21	3	8	1	15	2	0	32	34	251.77	326.04	1.30
Go	0	1	1	0	1	1	0	0	2	2	✗	✗	
HangOnMan	3	23	4	14	1	9	0	0	77	77	262.54	349.44	1.33
Mancala	3	34	4	33	1	1	0	0	73	73	8.26	11.03	1.34
MemoryGameOlderKids	15	27	15	27	0	0	0	0	50	50	659.42	804.57	1.22
Numeroo	2	17	3	15	1	2	0	0	73	73	1087.15	1383.90	1.27
Rabbit	6	18	7	10	1	8	0	0	101	101	1513.96	1849.72	1.22
ShoppingList	2	14	3	4	1	10	0	0	55	57	112.74	128.09	1.14
TodoList	0	42	1	27	1	15	0	0	90	90	1583.92	1787.17	1.13
BluetoothChat	1	4	1	0	0	4	0	12	9	50	7.37	27.61	3.75
CallLog	2	9	5	4	3	9	4	7	15	71	7.54	36.96	4.90
EventManager	1	5	5	1	4	5	1	6	11	53	7.23	26.07	3.61
AreaConverter	0	11	0	9	0	2	0	0	9	9	24.34	28.55	1.17
ComicsM	0	2	0	2	0	0	0	0	3	3	9.86	14.36	1.46
FullHouse	0	1	0	1	0	0	0	0	7	7	20.99	29.94	1.43
FusuiColorCompass	0	14	0	12	0	2	0	0	68	68	✗	416.59	
HandyPlanner	0	5	0	2	0	3	0	0	41	41	76.33	84.86	1.11
IM+	4	11	6	10	2	3	2	0	41	45	59.93	62.45	1.04
LoveCalculator	0	15	0	15	0	0	0	0	8	8	10.98	14.03	1.28
MahjongBook	6	18	7	17	1	1	0	0	100	115	193.94	241.37	1.24
SimpleCounter	2	1	2	1	0	0	0	0	30	30	43.90	61.22	1.39
UBRadioactive	0	6	17	17	17	6	17	3	19	129	26.91	✗	
Total	55	334	91	240	**36**	**120**	26	**28**	985	1257			

One limitation of the snapshot approach is that it is restricted by the information it stores. For example, we can store JavaScript function bodies but bodies of native functions. Also, dumping all possible information may incur large performance overhead. We may want to store information selectively just enough to reduce false positives.

4.2 Type-based Modeling for JS Libraries

Even though SAFE provides the most elaborate modeling for jQuery among existing open-source JavaScript analyzers [10, 23, 15], it cannot model complex behaviors precisely due to the inherent nature of static modeling. For example, consider the following jQuery uses in Bubblewrap:

```
$(this).addClass('popped')
       .css('background-image',
            poppedBubbleImages[Math.floor(Math.random() *
                               poppedBubbleImages.length)])
       .children().removeClass('bomb clock pump hammer');
```

Many jQuery library functions return jQuery objects as their results to execute function invocations continuously, and in the above example, the functions `addClass`, `css`, `children`, and `removeClass` all return jQuery objects as their results. Moreover, the input value of the `css` function is created with a random value, which is impossible to statically model precisely. This code is a typical example of generating false positives due to the modeling of JavaScript library APIs.

For such cases, we take return types of the corresponding library functions, generate arbitrary objects of the return types, and use them as return values. Inspired by automatic modeling of web APIs [6] specified in the Web Interface Description Language (WIDL) [3], we use type declarations of jQuery APIs in TypeScript [22] and manually model them in an on-demand fashion to reduce false positives.

4.3 Instrumentation for Dynamic File Loading

We collect "source code paths" dynamically to analyze dynamically loaded files. While dynamically loaded files may locate either in a local file system[13] or outside, we focus on only local files that are loaded dynamically in this paper.

To gather dynamically loaded file locations, we instrument all JavaScript files located in a local file system. When each of the JavaScript file evaluates, it logs its file location and we can collect the file locations of actually evaluated ones. Thus, local files that are loaded during the event loop phase are not evaluated, they do not remain any logs. Concretely, our instrumentor inserts `window.__dynamic_paths.push({PATH});` at the beginning of all the JavaScript files in a local file system. When each file evaluates, it saves the file path in the `__dynamic_paths` array, and we pass that information to SAFE to utilize that information in static analysis.

5. EVALUATION

This section evaluates our approach with the following 3 research questions:

- RQ1: **Precision**. What are the numbers of alarm changes after applying the techniques described in Section 4?
- RQ2: **Coverage**. What are the numbers of analyzed file and function changes after applying the techniques?
- RQ3: **Scalability**. What are the analysis time changes after applying the techniques?

To answer these questions, we use 17 cross-platform and 3 Tizen SDK, and 10 Tizen market applications. We manually investigated 389 alarms (Table 2) detected by the original SAFE bug detector and additional 63 alarms newly detected by applying our approach, identified true and false positives, and inspected what caused each of the false positives. We performed all experiments on a Mac OS X x64 machine with 3.5GHz Intel Core i7 CPU and 8GB Memory.

[13]By a local file system, we denote the directory that includes the main entry file like `index.html` and (recursively) its all subdirectories.

Table 5: Effects of dynamic information for precision

Causes	Subcategory	Before	After	Change	
		# FP	# FP	# RFP (Good)	# NFP (New)
W3C APIs	imprecise	22	19	3	0
	missing	138	74	70	6
Browser-specific APIs	imprecise	0	0	0	0
	missing	19	0	19	0
JavaScript library APIs	imprecise	35	44	0	9
	missing	1	2	0	1
Dynamic file loading		24	0	24	0
Dynamic code generation		6	6	0	0
Asynchronous calls		89	90	4	5
Others		0	4	0	4
Total		334	239	120	25

Table 6: Effects of logging dynamically loaded file paths

Application	# Total files	#Loaded files	
		Before	After
Flashcards	422	1	10
MemoryMatch	13	1	13
Tenframe	15	1	15
BluetoothChat	735	4	16
CallLog	730	4	11
EventManager	729	4	10
UBRadioactive	375	14	17

Table 4 summarizes the overall changes after applying our approach in terms of precision, coverage, and scalability. For presentation brevity, we show only 24 web applications that include alarms from user code.

Answer to RQ1.

For precision, we measure true and false positives before and after applying our approach, and compare changes in them. The results include 45 new alarms from the newly analyzed files by using dynamic information. Except for the same alarms reported before and after, our approach detected 36 more true positives (# NTP) and 26 more false positives (# NFP). Out of 334 false positives (Before # FP) before applying the mechanism, we could eliminate 120 alarms (# RFP) by using dynamic information (36%). Note that we did not miss any true positives. The overall analysis precision was improved from 14% (55/389) to 27% (91/331).

Table 5 shows the effects of using dynamic information in terms of precision for each cause of false positives. For each cause and subcategory, if any, the table presents the number false positives without and with dynamic information and changes in terms of reduced (for good) and new (for addional analysis coverage) false positives.

As we discussed in Section 4, we expect to see removed false positives thanks to execution environment snapshots for 2 characteristics: Browser-specific APIs and W3C APIs. Because snapshots provide dynamic information, the number of false positives about missing browser-specific APIs reduced 100% from 19 to 0. The snapshots could also provide W3C API information and the approach reduced the number of false positives for missing W3C APIs by 70. Among the remaining 74 missing W3C APIs, 73 are native functions and 1 requires extra modeling of CSS in an html document, which the snapshot approach cannot remove. Because we used existing modeling of W3C APIs rather than dumped information from snapshots, the number of false positives for imprecisely modeled W3C APIs did not change.

Answer to RQ2.

For coverage, we measure the numbers of added files and analyzed functions. Using dynamic information enabled the SAFE bug detector to analyze 25 more files, whose analysis reports 45 (24 true and 21 false) alarms that we did not denote in Table 4 for presentation brevity. The number of analyzed functions increased by 172 (from 985 to 1257). Note that analysis of 3 Tizen-SDK web applications covered more files using dynamic information, and their analysis also increased analyzed functions by 139. Analysis of Counting-Beads covered 2 more functions as well. As we discuss below, static analysis using dynamic information incurs some performance overhead, which results in unfinished analysis by timeout.

To understand the effects of analyzing only dynamically logged files in terms of coverage, we examined the changes in the number of analyzed files and compared them with the number of total files in each web application. Table 6 presents the numbers for 7 applications that load files dynamically. While our current approach cannot address dynamically loaded files from outside of local file systems, none of 30 target web applications used such patterns. While manually investigating dynamically loaded files, we observed that even though a single web application contains hundreds files, only a small fraction of them are actually used at run time. For example, Flashcards contains 422 JavaScript files, but 413 of them are located in the `lib/requirejs` directory and 409 of them are test-related code for `requirejs`. However, at run time, only a single library file `lib/requirejs/require.js` out of 413 files was used. Also, each of 3 Tizen SDK applications, BluetoothChat, CallLog, and EventManger, contains 2 directories `lib/tau` and `lib/tizen-web-ui-fw`, each of which contains 361 JavaScript files. Among 361 files, 353 files are for international locale settings. In our experiments, only `globalize.culture.ko.js` out of 353 files was dynamically loaded. Similarly, the UBRadioactive application contains the `tizen-web-ui-fw` directory, which contains 353 files are for international locale settings out of 359 JavaScript files.

Answer to RQ3.

For scalability, we compare the analysis time with the timeout of 2000 seconds. Out of 24 web applications, 3 did not finish analysis before using dynamic information; when using dynamic information, one more finished but two more did not finish analysis. Thus, we calculated the analysis time ratio for 4 web applications that finish analysis both without and with dynamic information. The analysis with dynamic information took 1.04 to 4.90 times longer; on average, the analysis time increased 1.64 times.

6. LESSONS LEARNED

While earlier work on static analysis of JavaScript web applications was based on some assumptions and observations that are appropriate for some class of web applications [12], we found that they may not be applicable to web applications in the wild:

- *"Through preliminary experiments we have found for the correctness properties that we focus on, the execution order of event handlers is often not crucial for the analysis precision."*

  As we discussed in Section 3.6, we observed that 89

alarms out of 344 false positives are due to asynchronous function calls.

- *"many of the dynamic features of JavaScript are not widely used in practice"*

  As we discussed in Sections 3.4 and 3.5, our manual investigation shows that 30 alarms out of 344 false positives are due to dynamic file loading and dynamic code generation.

- *"The study shows that the majority of method invocations in JavaScript are monomorphic."*

  The most frequently used JavaScript library, jQuery, provides a variety of overloaded functions: jQuery functions with the same name but with different numbers of arguments or different types of arguments are heavily used.

For our target applications, the major causes of false positives in the analysis results are W3C and browser-specific APIs. Even the state-of-the-art DOM modeling [25], which models frequently-used APIs based on extensive empirical studies, is not enough for analyzing real-world web applications. Also, manually modeling such APIs is tedious, error-prone, and labor intensive; it is almost impossible because different platforms provide different APIs and newer versions would invalidate existing modeling of older versions. We believe that a configurable, systematic, and automatic mechanism to model various environment like different execution environments with native function modeling is inevitable for analyzing real-world web applications.

Not only for native APIs like W3C and browser-specific APIs, but also JavaScript library APIs may get more benefits by modeling than actually analyzing them as target JavaScript programs. Even when JavaScript sources are available, modeling large and complex JavaScript libraries would take less analysis time by not analyzing them as analysis targets, produce more precise analysis results (depending on the precision of the modeling), and separate concerns of analyzing less interesting library code from analyzing more important user code. Using specific modeling may be unsound but we learned that developers prefer to improve precision with the expense of losing soundness in practice.

Among 7 kinds of causes, we could partially address 4 of them. The remaining causes are more challenging, and we plan to study them as our future work. To analyze dynamically generated code statically, we consider collecting concrete code objects by executing web applications multiple times, and use their abstract counterparts as "dynamically and automatically modeled code." Because a sound abstraction of dynamically generated code would be always the most imprecise value, we believe that the best practice would be a way to systematically tune the analysis precision by repetitive runs of concrete execution. Statically analyzing asynchronous function calls are similar to statically analyzing dynamically generated code in the sense that a sound abstraction of asynchronous calls would be always the most imprecise value, because any order of such calls would be possible. Finally, other causes are due to the analysis imprecision of the underlying analzer, SAFE. There must be rooms for improvement in the analysis precision of SAFE, but it is beyond the scope of this paper.

7. RELATED WORK

As JavaScript web applications become prevalent, several JavaScript static analyzers have been developed. TAJS [23, 13] is a type analysis for JavaScript programs, and it supports static analysis of web applications with a limited set of modeled APIs. To improve performance and scalability of JavaScript static analysis, it provides ad-hoc heuristics [5]. WALA [10, 29] was originally developed for static analysis of Java programs but now it is actively used in analysis of JavaScript and Android Java programs. The analysis precision of WALA is often less precise than other static analyzers for JavaScript, because it provides flow-insensitive analyses. SAFE [15, 18] is a general analysis framework for JavaScript including a default static analyzer based on abstract interpretation. While it provides several tools like an interpreter and a clone detector [7] for JavaScript, it has been mainly used for detecting type-related bugs using static analysis results [6]. Our approach may be applicable not only to SAFE but also for other static analyzers for JavaScript web applications.

Using both static and dynamic information for analyzing web applications is not a new idea. WALA [29] uses a dynamic determinacy analysis to identify determinate variables and expressions that hold the same values at all program points. Using such information, it can eliminate unreachable branches and specialize some code depending on determinate values. However, the analysis is too conservative that it often falls back to usual static analysis resulting in imprecise analysis results. JSBAF [30] constructs partial call graphs from dynamic execution traces from TracingSafari, an instrumented version of WebKit, and performs a static analysis using WALA. Because its call graphs do not contain unexecuted flows, it may not cover much of actual program flows. However, at the same time, because its call graphs include statically invisible code like dynamically generated and load code, they may cover missing flows from static analysis. While JSBAF uses the static analyzer of WALA on its unsound call graphs constructed dynamically, our mechanism uses dynamic information to aid sound analysis of sound call graphs.

Recording execution information at run time to use statically has been used for different purposes for programs written in different programming languages. Jsnap[14] dumps the state of a JavaScript program after executing the top-level code, but before the event loop. TSCheck [9][15] uses jsnap to find bugs in hand-written TypeScript type definitions by comparing them with their corresponding JavaScript libraries. It uses dumped states to skip analysis of code at loading time but to focus on analyzing code after loading. On the contrary, we take snapshots of execution environments even before loading any code, which contain no user-related information but pure environment information. Also, while we dump most information including non-native function bodies, jsnap dumps only such information required to check TypeScript type definitions which does not include function bodies. Phantm [17, 16], a static analyzer for PHP programs uses a similar approach. It instruments PHP programs to dump their information during evaluation, which later Phantm uses to improve its analysis precision.

[14]https://github.com/asgerf/jsnap
[15]https://github.com/asgerf/tscheck

8. CONCLUSION

Built on top of recent research achievements in static analysis of JavaScript web applications, we perform a concrete, realistic case study on reducing false positives in static analysis of web applications. With 30 web applications including market applications and using the state-of-the-art JavaScript static analyzer, SAFE, we performed empirical studies on causes of false positive in static analysis results. By manually investigating all the alarms from target applications, we identified 7 causes: W3C APIs, browser-specific APIs, JavaScript library APIs, dynamic file loading, dynamic code generation, asynchronous function calls, and others. From the observation that some causes may be easily lessened by utilizing specific information aggressively with the expense of unsoundness, we developed a mechanism to use dynamic information like snapshots of execution environments and logs of dynamically loaded files, and type-based modeling. Our experimental evaluation shows that with dynamic snapshots of execution environments, we could eliminate 70 out of 138 and all 19 false positives for W3C APIs and browser-specific APIs, respectively. In addition, by including dynamically loaded files in static analysis, we eliminated all 24 false positives due to dynamic file loading. While our mechanism incurs 1.64x performance overhead on average, it did not miss any true positives, removed many false positives, detected more true positives, and increased the number of analyzed functions.

9. ACKNOWLEDGMENTS

We thank Sungho Lee, Yoonseok Ko, and the members of PLRG@KAIST for their contributions in implementing the proposed methodologies. This work is supported in part by Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grants NRF-2014R1A2A2A01003235) and Samsung Electronics.

10. REFERENCES

[1] ECMAScript Language Specification. Edition 5.1. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[2] HTML5: A vocabulary and associated APIs for HTML and XHTML. http://www.w3.org/TR/html5/.

[3] Web IDL. http://www.w3.org/TR/WebIDL.

[4] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript event-based interactions. In *ICSE 2014*.

[5] E. Andreasen and A. Møller. Determinacy in static analysis for jQuery. In *OOPSLA 2014*.

[6] S. Bae, H. Cho, I. Lim, and S. Ryu. SAFE$_{\mathsf{WAPI}}$: Web API misuse detector for web applications. In *FSE 2014*.

[7] W. T. Cheung, S. Ryu, and S. Kim. Development nature matters: An empirical study of code clones in JavaScript applications. *Empirical Software Engineering*, 2015.

[8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*.

[9] A. Feldthaus and A. Møller. Checking correctness of TypeScript interfaces for JavaScript libraries. In *OOPSLA 2014*.

[10] IBM Research. T.J. Watson Libraries for Analysis (WALA). http://wala.sf.net.

[11] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *ISSTA 2012*.

[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 2011*.

[13] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *SAS 2009*.

[14] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE 2013*.

[15] KAIST PLRG. SAFE: Scalable Analysis Framework for ECMAScript. http://safe.kaist.ac.kr, 2013.

[16] E. Kneuss, P. Suter, and V. Kuncak. Phantm: PHP analyzer for type mismatch (research demonstration). In *FSE 2010*.

[17] E. Kneuss, P. Suter, and V. Kuncak. Phantm: PHP analyzer for type mismatch. http://lara.epfl.ch/w/phantm, 2012.

[18] 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 2012*.

[19] B. S. Lerner, M. J. Carroll, D. P. Kimmel, H. Q.-D. La Vallee, and S. Krishnamurthi. Modeling and reasoning about DOM events. In *Proceedings of the 3rd USENIX Conference on Web Application Development*, 2012.

[20] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 2015.

[21] F. Meawad, G. Richards, F. Morandat, and J. Vitek. Eval begone!: Semi-automated removal of eval from JavaScript programs. In *OOPSLA '12*.

[22] Microsoft. TypeScript. http://www.typescriptlang.org.

[23] A. Møller, S. H. Jensen, P. Thiemann, M. Madsen, M. D. Ingesman, P. Jonsson, and E. Andreasen. TAJS: Type analyzer for JavaScript. https://github.com/cs-au-dk/TAJS, 2014.

[24] C. Park and S. Ryu. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *ECOOP 2015*.

[25] C. Park, S. Won, J. Jin, and S. Ryu. Static analysis of JavaScript web applications in the wild via practical DOM modeling. In *ASE 2015*.

[26] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: a large-scale study of the use of eval in JavaScript applications. In *ECOOP 2011*.

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

[28] C. Sadowski, J. van Gogh, C. Jaspan, E. Soederberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *ICSE 2015*.

[29] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Dynamic determinacy analysis. In *PLDI 2013*.

[30] S. Wei and B. G. Ryder. Practical blended taint analysis for JavaScript. In *ISSTA 2013*.