

Toward Analysis and Bug Finding in JavaScript Web Applications in the Wild

Sukyong Ryu, Jihyeok Park, and Joonyoung Park,
Korea Advanced Institute of Science and Technology

// We present our journey to analyze and find bugs in JavaScript web applications in the wild. We describe technical challenges in analyzing them and our solutions to address the challenges via a series of open source analysis frameworks, the scalable analysis framework for ECMAScript (SAFE) family. //



JavaScript Web Applications in the Wild

JavaScript was first developed as a simple scripting language, and, now, it is the de facto standard language for web programming. It is highly expressive

thanks to function values and dynamic code generation; JavaScript functions can take callback functions as their arguments, and they can generate code to execute during evaluation. Also, JavaScript is extremely portable. It can run on any device, such as smart TVs or smart watches, without installation of a compiler or

an interpreter if the devices have a browser.

However, the expressivity and portability of JavaScript also introduce various errors and vulnerabilities, collectively known as *bugs*, in JavaScript web applications. Unlike statically typed languages, such as C and Java, JavaScript does not prevent type-related errors, such as calling functions with wrong numbers or wrong types of arguments, which results in frequent type-related errors by developers. In addition, because JavaScript web applications often use third-party libraries on browsers, they are vulnerable to security attacks.

Our long-term goal is to develop a tool that can analyze and detect bugs in real-world JavaScript web applications. This article reports on our recent significant progress toward the goal, the SAFE family illustrated in Figure 1. We put the series of efforts into context, providing a big picture of the JavaScript analysis for practitioners. In the rest of this article, we incrementally present each of the technical challenges and their proposed solutions, summarized in Table 1 using the SAFE family, share our experiences in both academia and industry, and discuss the current limitations and the path forward.

Analysis of JavaScript Programs

JavaScript has various characteristics that make program analysis especially difficult. It does not have a compile-time type system. A JavaScript variable may have a value of any six types—undefined, null, Boolean, number, string, and object—at any point of program evaluation, and it may be implicitly converted to different types depending on its surrounding contexts, according to the

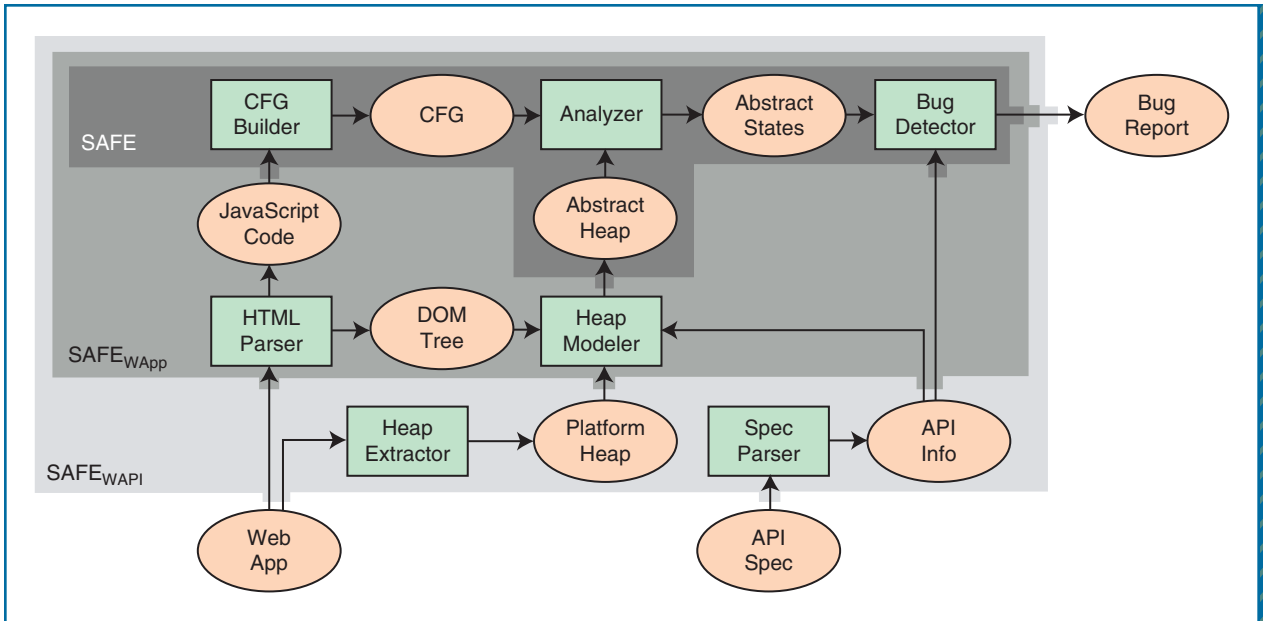


FIGURE 1. Architectural overview of the SAFE family. API: application programming interface. DOM: Document Object Model.

language specification. Such behaviors give much freedom to JavaScript developers, but at the same time, they often introduce type-related errors. JavaScript provides multiple ways to generate code at runtime; several functions, such as `eval`, take string arguments that represent JavaScript code, realize them at runtime, and run them. Because such dynamically generated code fragments are not available at compile time, they are not in the targets of static analysis. In addition, JavaScript is not statically scoped; one language construct, the `with` statement, introduces scopes dynamically, which is a challenge for static analysis. Because JavaScript programs use various libraries, such as jQuery, which use loops and first-class property names extensively, precise analysis of loops and string values becomes critical.

Let us consider the JavaScript code in Figure 2. The `with` statement on lines 7–9 shows that all of the properties in the object `o` (line 7)

Table 1. An analysis of real-world JavaScript web applications via the SAFE family.

Analysis Targets	Technical Challenges	Analysis Techniques
JavaScript programs	Dynamic code generation Dynamic scoping via <code>with</code> statements Join of analysis results for loops First-class property names	Rewriting simple code with rewriting Loop-sensitive analysis Regular expression domain
Web applications	DOM objects and APIs Dynamic file loading	Modeling DOM objects and APIs Analysis with dynamic browser information Analysis with dynamically loaded files
Hybrid applications	Platform APIs	Modeling platform APIs API misuse detection

become local variables within the body of the statement (line 8); thus, the names `a`, `b`, and `c` are dynamically introduced within the body of the statement. The `eval` function call on line 12 shows that the string literal `'o[name] = o[name]();'` becomes JavaScript code to evaluate at runtime.

Also, the `for-in` loop on lines 11–13 reveals that the property names of the object `o` are first-class values that are generated by iteration on the properties of `o`.

To address the challenges in statically analyzing JavaScript programs, we developed an open source analysis

```

1  /**** Analysis of JavaScript Programs *****/
2
3  function f() { return 0; }
4  function g() { return 1; }
5  function h() { return 2; }
6  var o = { a : 0, b : 1, c : 2 };
7  with(o) {
8      a = f; b = g; c = h;
9  };
10
11 for (name in o) {
12     eval('o[name] = o[name]();');
13 }
14
15 /**** Analysis of JavaScript Web Applications *****/
16
17 window.requestAnimationFrame =
18     window.requestAnimationFrame ||
19     window.webkitRequestAnimationFrame || // Chrome, Opera
20     window.mozRequestAnimationFrame || // FireFox
21     window.oRequestAnimationFrame || // Old Opera
22     window.msRequestAnimationFrame || // IE
23     function (callback) {
24         use strict;
25         window.setTimeout(callback, 1000 / 60);
26     };
27
28 function isDiv(elem) {
29     // wrong usage of '==='
30     return elem.tagName.match(/^\w+/) === 'DIV';
31 }
32
33 /**** Analysis of JavaScript Hybrid Applications *****/
34
35 function calendarListCallback(calendars) {
36     calendars.map(calendar => calendar.type);
37 }
38
39 webapis.calendar.getCalendars("EVENT", calendarListCallback);

```

FIGURE 2. A running example named `sample.js` for analysis of real-world JavaScript web applications.

framework for JavaScript—SAFE.¹ The innermost box in Figure 1 illustrates an architectural overview of SAFE. It takes a JavaScript program and processes it like a conventional program analysis framework does. Among others, CFG Builder rewrites `eval`-like function calls that have string literal arguments into JavaScript code that the arguments represent, and it replaces most `with` statements with other language constructs in a sound manner.² The Analyzer performs a static analysis based on the abstract

interpretation framework using abstract values that can represent all six kinds of JavaScript value types, and it provides loop-sensitive analysis³ and regular expression domains for string analysis.⁴ Thus, it can analyze the code in Figure 2 even in the presence of the `with` statement, the `eval` function call, and the `for-in` loop using first-class property names. Finally, Bug Detector uses the analysis results to find type-related bugs, such as undefined variable accesses that cause the `ReferenceError` exception in JavaScript and

function calls with nonfunction values that cause the `TypeError` exception. Note that SAFE provides “soundy” analysis.⁵ It cannot analyze `eval` function calls with arguments that are not string literals, for example.

Analysis of JavaScript Web Applications

The first step toward analysis of real-world JavaScript web applications is to analyze JavaScript code embedded in HTML documents. JavaScript programs are often executed in

host environments, and the most widely used host environments are web browsers. HTML documents of webpages or web applications may contain JavaScript code fragments; web browsers represent the HTML document structures as host objects called *Document Object Model (DOM)*, and DOM application programming interfaces (APIs) provide ways for JavaScript code to use host objects. Thus, the analyzing of JavaScript web applications requires understanding of not only the JavaScript semantics but also the meaning of DOM structures and the interactions between DOM and JavaScript. Moreover, web applications often load JavaScript files dynamically, and JavaScript code fragments running on various browsers use browser-specific APIs, resulting in static analysis imprecision. In addition, JavaScript web applications interact with users via event handling functions. JavaScript event functions may be registered to HTML documents as HTML attributes, such as `onload` or `onclick`, before program execution, and they may be registered via functions, such as `addEventListener`, during program execution. Then, users can make JavaScript event functions called asynchronously. Similarly, for dynamically generated JavaScript code and dynamically loaded files, event function calls that are invoked by user inputs are not available before program execution. Therefore, purely static analysis will miss them from the analysis targets, which results in unsound analysis results. To analyze such dynamic event behaviors soundly, most existing static analyzers assume that any event functions may be called in any order.

Consider the code example in Figure 2 again. Because different

browsers may use different API functions with similar functionalities, if the global function `requestAnimationFrame` (line 17) is not yet initialized (line 18), it is initialized to one of the browser-specific API functions (lines 19–22) or a default function (lines 23–26). Although the code supports browser compatibility between different versions and different browsers, most static analyzers lose precision by considering all possible cases to analyze them soundly. Then, it defines a function named `isDiv` (lines 28–31), which takes a DOM element and checks whether its tag name is 'DIV'. The function `isDiv` is invoked when a user clicks the DOM elements on lines 3–4 in Figure 3. The function calls on lines 4 and 5 evaluate to true and false, respectively.

Note that as the comment on line 29 of Figure 2 states, the function `isDiv` contains a bug. Although the value of `elem.tagName.match(/^\w+/)` is either an object or `null`, the value of `DIV` is 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. Actually, this code is a simplified version of the Wikipedia webpage. When a user clicks the language selection button from the Wikipedia page, a JavaScript function named `setLang` is called, which contains the bug that `isDiv` replicates. We reported the bug to the Wikipedia developers, and they confirmed the

bug and fixed it right away by replacing `===` with `==`. Because various Wikimedia projects share the code, the same bug existed in many webpages. A funny thing is that the bug revived in less than a year! We found the same bug again while evaluating yet another analysis technique against the Wikipedia page and reported it again. What happened is that a developer revised some code unrelated to `setLang`, but a simple syntactic checker, JSHint, warned the developer to replace `==` with `===` because `===` is considered to be better. After inspecting the situation, the Wikipedia developers revised the code in another way using `===` correctly.

Thus, to analyze JavaScript web applications and to detect bugs in them, we developed `SAFEWApp` by extending `SAFE` with DOM modeling and analysis with various dynamic information. The middle box in Figure 1 shows our extensions on `SAFE`. In addition to analyzing JavaScript programs, `SAFEWApp` can analyze web applications; it takes HTML documents as inputs, extracts embedded JavaScript code fragments, builds their CFGs using `SAFE`, constructs DOM models, analyzes them collectively using the Analyzer of `SAFE`, and detects type-related bugs via the Bug Detector. `SAFEWApp` supports DOM APIs by providing faithful (partial) DOM models.⁶ To address event functions that handle user inputs, it also builds CFGs for event

```

1 <html>
2   <head> <script src="sample.js"></script> </head>
3   <body> <div onclick="isDiv(this)">foo</div>
4         <p  onclick="isDiv(this)">bar</p> </body>
5 </html>

```

FIGURE 3. The HTML code that interacts with the JavaScript code in Figure 2.

```

1 [NoInterfaceObject]
2 interface CalendarManager {
3     void getCalendars(CalendarType type,
4                       CalendarArraySuccessCallback successCallback,
5                       optional ErrorCallback? errorCallback
6                       ) raises(WebAPIException);
7     ...
8 };
9
10 [Callback=FunctionOnly, NoInterfaceObject]
11 interface CalendarArraySuccessCallback {
12     void onsuccess(Calendar[] calendars);
13 };
14
15 [NoInterfaceObject]
16 interface Calendar {
17     readonly attribute CalendarId id;
18     readonly attribute DOMString name;
19     ...
20 };

```

FIGURE 4. The specification of a native function `getCalendars` called by the JavaScript code in Figure 2.

functions and passes them together with JavaScript CFGs to the Analyzer in SAFE. $\text{SAFE}_{\text{WApp}}$ utilizes dynamic information in two ways.⁷ First, it collects dynamic browser information, such as concrete values of built-in properties, to specialize the Analyzer. For web applications running on a specific browser, such as Chrome, the value of the function `requestAnimationFrame` (line 17) in Figure 2 definitely becomes `window.webkitRequestAnimationFrame`, which, in turn, improves the analysis precision. Second, $\text{SAFE}_{\text{WApp}}$ collects dynamically loaded files and includes them as its analysis targets. To collect such dynamic information, it instruments input JavaScript web applications so that they can log necessary information during execution, and it runs the instrumented web applications.

Analysis of JavaScript Hybrid Applications

The prevalence of the HTML5 technologies widely expands the realm of JavaScript applications to the Internet

of Things and smart appliances, such as smart TVs and smart watches, which introduces new kinds of challenges in static analysis. To utilize device-specific features, JavaScript web applications call native library functions provided by device platforms via their APIs. For example, JavaScript code in an Android application running on an Android phone may invoke native methods written in Android Java to access its geographic location. Thus, analyzing the behaviors of such JavaScript applications asks for analysis of function call flows between software components written in different programming languages.

Also, native functions often take callback functions to communicate with the application code. For example, Figure 4 shows partial specifications of web APIs for calendar functionalities. The `CalendarManager` interface contains several functions, including `getCalendars`, and the `Calendar` interface contains two attributes, `id` and `name`, and more. The `getCalendars` function takes two or three arguments where the second argument

denotes a callback function to be called by the native code if the execution of the function succeeds, and the third optional argument denotes a callback function to be called otherwise. Figure 2 presents how the native function may be called from JavaScript code on line 39; the second argument is the function defined on lines 35–37. Although calls of the callback function are syntactically invisible, the function may be called by `getCalendars`. Therefore, to detect the bug on line 36, which accesses the absent `type` attribute of a calendar, analysis of JavaScript web applications requires understanding of such implicit call behaviors.

Hence, we extended $\text{SAFE}_{\text{WApp}}$ to analyze both explicit and implicit interaction flows between JavaScript and native code by providing automatic modeling of API functions. We observed that vendors specify the API functions they provide in some specification languages, such as Web IDL and TypeScript, so that third-party developers can build web applications for their devices. Although the JavaScript semantics are wildly dynamic, JavaScript code interacting with native code should satisfy the API specifications to behave correctly. Based on this observation, we developed $\text{SAFE}_{\text{WAPI}}$,⁸ which takes the API specifications of native functions and builds their abstract models automatically, so that the main Analyzer can understand the behaviors of the API functions. The outermost box in Figure 1 illustrates that Spec Parser builds API Info from API Specs, and Heap Modeler integrates them with DOM models generated from input applications and passes the integrated abstract model to the Analyzer. Because the approach is parameterized by specification languages, $\text{SAFE}_{\text{WAPI}}$ can support any



RELATED WORK IN THE ANALYSIS TOOL OF JAVASCRIPT APPLICATIONS

Since its introduction in 1995, the JavaScript programming language has been used widely in industry, and the research community became interested in JavaScript since 2005. Here, we briefly survey static and dynamic analysis tools of JavaScript applications.

STATIC ANALYSIS TOOLS OF JAVASCRIPT APPLICATIONS

In addition to the SAFE family, several open source tools can analyze JavaScript applications statically while focusing on different purposes for their analyses. Type Analyzer for JavaScript (TAJS)^{S1} is the closest one to the SAFE family. Although the SAFE family analyzes more real-world web applications by identifying and resolving concrete issues in the wild, TAJS has been focusing on devising static analysis techniques that target problems related to the JavaScript semantics. T.J. Watson Libraries for Analysis (WALA)^{S2} was originally developed for Java bytecode analysis, and now it supports JavaScript analysis as well. After attempting several sound static analysis techniques for JavaScript,^{S2, S3} WALA is leaning toward unsound call graph construction to utilize its efficiency.^{S4} JavaScript Abstract Interpreter (JSAI)^{S5} is a JavaScript analysis framework that supports easy configuration of analysis sensitivities. Unlike the SAFE family, TAJS, and WALA, JSAI does not construct control flow graphs explicitly. JSAI is based on the abstract machine semantics, whereas the other tools are based on the big-step operational semantics.

DYNAMIC ANALYSIS TOOLS OF JAVASCRIPT APPLICATIONS

Jalangi^{S6} is an open source dynamic analysis framework for JavaScript. It provides simple dynamic analyzers, such as concolic testing and dynamic taint analysis, as sample uses of Jalangi, and it has been actively used in several dynamic analyses.^{S7} Also, dynamic analyzers can detect event-related errors. WebRacer^{S8} is the first dynamic race detector for web applications, EventRacer^{S9} can detect more severe bugs than WebRacer can, with fewer false positives, and R4^{S10} can distinguish harmful races from the races reported by EventRacer by using a stateless model checker for event-driven applications.

References

- S1. S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *Proc. Int. Symp. on Static Analysis*, 2009, pp. 238–255.
- S2. M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, "Correlation tracking for points-to analysis of JavaScript," in *Proc. Eur. Conf. Object-Oriented Programming*, 2012, pp. 435–458.
- S3. M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Dynamic determinacy analysis," in *Proc. 34th ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2013, pp. 165–174.
- S4. A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for JavaScript IDE services," in *Proc. 2013 Int. Conf. Software Eng.*, pp. 752–761.
- S5. V. Kashyap et al., "JSAI: A static analysis platform for JavaScript," in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations Software Eng.*, 2014, pp. 121–132.
- S6. K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A tool framework for concolic testing, selective record-replay, and dynamic analysis of JavaScript," in *Proc. Int. Symp. Foundations Software Eng.*, 2013, pp. 615–618.
- S7. M. Pradel, P. Schuh, and K. Sen, "TypeDevil: Dynamic type inconsistency analysis for JavaScript," in *Proc. 37th Int. Conf. Software Eng.*, 2015, pp. 314–324.
- S8. B. Petrov, M. Vechev, M. Sridharan, and J. Dolby, "Race detection for web applications," in *Proc. 33rd ACM SIGPLAN Conf. Programming Language Des. and Implementation*, 2012, pp. 251–262.
- S9. V. Raychev, M. Vechev, and M. Sridharan, "Effective race detection for event-driven programs," in *Proc. 2013 ACM SIGPLAN Conf. Object-Oriented Programming, Syst., Languages, and Appl.*, pp. 151–166.
- S10. C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. Vechev, "Stateless model checking of event-driven applications," in *Proc. 2015 ACM SIGPLAN Conf. Object-Oriented Programming, Syst., Languages, and Appl.*, pp. 57–73.

RELATED WORK IN THE ANALYSIS OF WEB APPLICATIONS

Expanding analysis targets, researchers have studied analysis of Document Object Model (DOM)-related behaviors and various kinds of web applications.

ANALYSIS OF DOM-RELATED BEHAVIORS

Jensen et al.^{S11} used overapproximation of DOM tree structure and event flows. They analyzed events by considering all possible combinations of event calls, which is sound but which also makes the analysis results overapproximated. By using dynamic information, SAFE_{WApp} can improve the analysis precision. Alimadadi et al.^{S12, S13} developed various tools to help developers understand complex dynamic behaviors of web applications. They built Clematis,^{S12} which captures low-level event-based interactions in web applications and transforms the information to high-level representations that developers can understand. To provide DOM-sensitive change impact analysis for JavaScript, they developed Tochal,^{S13} which builds static dependency graphs augmented with dynamic information for DOM-sensitive changes. Their tools aim to help developers' in understanding program behaviors rather than finding bugs.

ANALYSIS OF WEB APPLICATIONS BEYOND JAVASCRIPT

Nguyen et al.^{S14} studied usage patterns in JavaScript web applications by proposing two approaches: JSModel represents JavaScript usages as graphs, and JSMiner mines interprocedural, data-oriented JavaScript usage patterns. They also built HTML/JavaScript variability-aware parsers so that IDEs can build call graphs for embedded JavaScript code in HTML documents.^{S15} Using the parsers, they produced WebSlice, which performs program slicing across different languages for web applications.^{S16} Ocariza et al.^{S18} developed Aurebesh, a tool that detects type inconsistencies in AngularJS^{S17} applications, which is the most popular JavaScript MVC framework. Feldthaus and Møller^{S19} developed TSCheck, a tool that detects inconsistencies between TypeScript type declaration files and their corresponding JavaScript library implementation. Finally, Artzi et al.^{S20} studied fault localization for Hypertext Preprocessor web applications.

References

- S11. S. H. Jensen, M. Madsen, and A. Møller, "Modeling the HTML DOM and browser API in static analysis of JavaScript web applications," in *Proc. 19th ACM SIGSOFT Symp. and 13th Eur. Conf. Foundations Software Eng.*, 2011, pp. 59–69.
- S12. S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, "Understanding JavaScript event-based interactions," in *Proc. 36th Int. Conf. Software Eng.*, 2014, pp. 367–377.
- S13. S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Hybrid DOM-sensitive change impact analysis for JavaScript," in *Proc. Eur. Conf. Object-Oriented Programming*, 2015, pp. 321–345.
- S14. H. V. Nguyen, H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Mining interprocedural, data-oriented usage patterns in JavaScript web applications," in *Proc. Int. Conf. Software Eng.*, 2014, pp. 791–802.
- S15. H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Building call graphs for embedded client-side code in dynamic web applications," in *Proc. Int. Symp. Foundations of Software Eng.*, 2014, pp. 518–529.
- S16. H. V. Nguyen, C. Kaestner, and T. N. Nguyen, "Cross-language program slicing for dynamic web applications," in *Proc. Int. Symp. Foundations Software Eng.*, 2015, pp. 369–380.
- S17. AngularJS, 2010. Accessed on: Feb. 1, 2019. [Online]. Available: <http://www.angularjs.org>
- S18. F. Ocariza, K. Pattabiraman, and A. Mesbah, "Detecting inconsistencies in JavaScript MVC applications," in *Proc. Int. Conf. Software Eng.*, 2015, pp. 325–335.
- S19. A. Feldthaus and A. Møller, "Checking correctness of TypeScript interfaces for JavaScript libraries," in *Proc. Conf. Object-Oriented Programming, Syst., Languages, and Appl.*, 2014, pp. 1–16.
- S20. S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Practical fault localization for dynamic web applications," in *Proc. Int. Conf. Software Engineering*, 2010, pp. 265–274.

native APIs if their specifications provide their names and types. Although the mechanism cannot model impure behaviors of native functions, such as input–output and mutation of global variables, it can model type-level features, such as numbers and types of arguments, return types, and checked exceptions.

Current Status of the SAFE Family


The SAFE family is publicly available,⁹ and it has been used in both academia and industry. Various components of SAFE are being used in academia, such as KJS, a complete formal semantics of JavaScript,¹⁰ and JavaScript Abstract Interpreter, a JavaScript static analysis framework.¹¹ SAFE was also used to formalize and implement a JavaScript module system,¹² whose formalization revealed inconsistent behaviors between different implementations from Google and Microsoft.¹³ Beyond academia, SAFE has been applied to several projects in industry, such as those with Samsung, Oracle, and IBM. SAFE is integrated in the software development kit of Tizen, an open source software platform for multiple devices, which is a Linux Foundation project.

As stated previously, because SAFE provides “soundy” analysis, it cannot soundly analyze JavaScript applications that use getters and setters, `eval` function calls with arguments that are not string literals, dynamically available code, and frameworks that are not fully modeled in SAFE, among others. For example, although SAFE supports analysis of jQuery, it also provides partial modeling of jQuery to exclude jQuery from analysis targets while losing soundness. By sacrificing soundness, the SAFE family provides precise analysis results. Using $\text{SAFE}_{\text{WAPI}}$,⁸ which can detect misuse of native API functions,

we found that four TV applications and one mobile application out of 43 applications access absent properties of platform objects. Because $\text{SAFE}_{\text{WAPI}}$ reported misuse cases precisely, developers fixed 80 bug reports out of 88, which amounts to a 91% fix rate.

Currently, the main challenge of SAFE is the analysis scalability. Although most web applications analyzed in academia have fewer than 10,000 lines of JavaScript code (LOC), real-world web applications in industry have about 700,000 LOC in Oracle and millions in Facebook. In attempts to analyze real-world web applications, the SAFE family has tried various approaches to reduce the amount of computation. For example, because SAFE focuses on a specific browser using dynamic information instead of considering all of the browsers, it reduces the amount of execution flows to analyze, which in turn improves the analysis precision. In addition, by supplementing partial modeling with dynamic information, SAFE analyzes previously unreachable flows, which enlarges the coverage of analysis targets. We evaluated the effects of dynamic information,⁷ and the experimental results showed that the analysis precision became about twice as good, although the analysis time increased due to newly analyzed targets. Finally, we have been integrating the unsound but efficient T.J. Watson Libraries for Analysis (WALA) analysis with SAFE by selecting analysis targets via WALA and then analyzing them via $\text{SAFE}_{\text{WApp}}$,¹⁴ and we also analyze flows between JavaScript and Android Java by extending WALA.¹⁵

Analysis of JavaScript web applications is one of the most active research areas: see “Related Work in the Analysis Tool

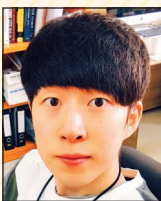
of JavaScript Applications,” which summarizes research on static and dynamic analysis tools of JavaScript applications, and “Related Work in the Analysis of JavaScript Web Applications,” which discusses analysis of various web applications. However, ample open challenges still remain. First, analyzing information that is available only at runtime is one of the fundamental challenges. More aggressive use of dynamic information than the current approaches for dynamic code generation and dynamic event function calls is a promising direction. Second, one of the practical challenges is the analysis of frameworks and APIs implemented in different languages. On top of many frameworks, such as Node and Cordova, Samsung and Oracle have their own in-house frameworks written in platform-specific languages. To handle them without actually analyzing their source code, a systematic mechanism that models their behaviors would open up new horizons, as would sharing and merging various models developed for different analyzers. Finally, finding the best configuration for various analysis techniques supported by analyzers is a difficult task. To help practitioners get precise analysis results in a scalable manner without too much knowledge of the analyzer details, advanced mechanisms that automatically configure analysis techniques would be extremely useful. We believe that analysis and bug detection of JavaScript web applications is becoming more important, and the research community is moving forward. 

Acknowledgments

We thank Changhee Park, Hongki Lee, SungGyeong Bae, Sora Bae, and Yeonhee Ryou for their helpful comments. We appreciate the members of the Programming Language



SUKYOUNG RYU is an associate professor in the School of Computing at Korea Advanced Institute of Science and Technology (KAIST). Her research interests include programming languages, program analysis, and programming environment. Ryu received a Ph.D. in computer science from KAIST. She is a Member of the IEEE and ACM. Contact her at sryu.cs@kaist.ac.kr.



JIHYEOK PARK is a Ph.D. student in the School of Computing at Korea Advanced Institute of Science and Technology (KAIST). His research interests include static analysis, memory abstraction, and formal verification. Park received a B.S. in computer science from KAIST. Contact him at jhpark0223@kaist.ac.kr.



JOONYOUNG PARK is a Ph.D. student in the School of Computing at Korea Advanced Institute of Science and Technology (KAIST). His research interests include dynamic analysis, bug detection of web applications, and concolic testing. Park received an M.S. in computer science from KAIST. Contact him at gmb55@kaist.ac.kr.

Research Group at Korea Advanced Institute of Science and Technology and our colleagues at S-Core and Samsung Electronics who made substantial contributions to the development of the SAFE family. This work is supported in part by the National Research Foundation of Korea (grant NRF-2017R1A2B3012020), the Institute for Information and Communications Technology Promotion (grant R1707711), and Samsung Electronics.

References

1. H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript," in *Proc. Int. Workshop Foundations Object-Oriented Languages*, 2012.
2. C. Park, H. Lee, and S. Ryu, "All about the with statement in JavaScript: Removing with statements in JavaScript applications," in *Proc. Symp. Dynamic Languages*, 2013, pp. 73–84.
3. C. Park and S. Ryu, "Scalable and precise static analysis of JavaScript applications via loop-sensitivity," in *Proc. Eur. Conf. Object-Oriented Programming*, 2015, pp. 735–756.
4. C. Park, H. Im, and S. Ryu, "Precise and scalable static analysis of jQuery using a regular expression domain," in *Proc. Symp. Dynamic Languages*, 2016, pp. 25–36.
5. B. Livshits et al., "In defense of soundness: A manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015. [Online]. Available: <https://dl.acm.org/citation.cfm?doid=2728770.2644805>
6. C. Park, S. Won, J. Jin, and S. Ryu, "Static analysis of JavaScript web applications in the wild via practical DOM modeling," in *Proc. Int. Conf. Automated Software Eng.*, 2015, pp. 552–562.
7. J. Park, I. Lim, and S. Ryu, "Battles with false positives in static analysis of JavaScript web applications in the wild," in *Proc. Int. Conf. Software Eng.*, 2016, pp. 61–70.
8. S. Bae, H. Cho, I. Lim, and S. Ryu, "SAFE_{WAPI}: Web API misuse detector for web applications," in *Proc. Int. Symp. Foundations of Software Eng.*, 2014, pp. 507–517.
9. S. Ryu, "Scalable Analysis Framework for ECMAScript," GitHub, 2016. [Online]. Available: <https://github.com/sukyounge/safe>
10. D. Park, A. Ştefănescu, and G. Roşu, "KJS: A complete formal semantics of JavaScript," in *Proc. Conf. Programming Language Des. and Implementation*, 2015, pp. 428–438.
11. V. Kashyap et al., "JSAI: A static analysis platform for JavaScript," in *Proc. Int. Symp. Foundations Software Eng.*, 2014, pp. 121–132.
12. S. Kang and S. Ryu, "Formal specification of a JavaScript module system," in *Proc. Conf. Object-Oriented Programming, Syst., Languages, and Appl.*, 2012, pp. 621–638.
13. J. Cho and S. Ryu, "JavaScript module system: Exploring the design space," in *Proc. 13th Int. Conf. Modularity*, 2012, pp. 229–240.
14. Y. Ko, H. Lee, J. Dolby, and S. Ryu, "Practically tunable static analysis framework for large-scale JavaScript applications," in *Proc. Int. Conf. Automated Software Eng.*, 2015, pp. 541–551.
15. S. Lee, J. Dolby, and S. Ryu, "HybridDroid: Static analysis framework for Android hybrid applications," in *Proc. Int. Conf. Automated Software Eng.*, 2016.