S-Core

April 2018 | Ver. 1.0

DeepScan

Detecting JavaScript Errors and Code Smells with Static Analysis

Contents

JavaScript is the most commonly used programming language and has been at the top of the GitHub annual highlights. Its open source ecosystem will be much bigger as it is widely applied to developing server, mobile, and desktop applications beyond just websites.

While code bases written in JavaScript are larger, debugging and managing the code bases are getting more difficult especially due to the dynamic features of JavaScript language. But you can significantly reduce this quality cost by catching code defects at the earlier stage of the development cycle.

Static analysis tools come into play at this stage.

To help JavaScript folks find static analysis tools helpful, we rummaged our database regarding run-time errors and code smells found by our JavaScript static analysis tool. It was gathered by analyzing code from thousands of public JavaScript and TypeScript projects on GitHub.

This report describes how static analysis tools work and shows some types of problems that static analysis tools can prevent.

0. Executive Summary

DeepScan is a static analysis tool designed to help organizations achieve better code quality for the release of applications written in JavaScript. We engineered it to detect code defects more precisely and have collected code defects from thousands of open source projects on GitHub.

This report provides some of the ways static analysis tools can address the troubles arrived with an increasing use of JavaScript. Sections will describe how static analysis tools work and show examples of errors and poor code quality collected.

The results offer a number of findings:

- Static analysis tools can detect code defects that linters cannot.
- By adopting static analysis tools, you can prevent run-time errors and poor code quality at the earlier stage of the development cycle.
- Static analysis tools can help JavaScript developers and testers upgrade their language skills by directly educating them about questionable coding practices.
- With consistency and faster speed, automated static analysis tools can detect code defects human developers might have missed.

1. Introduction

JavaScript to date has become popular language in the world and it, more especially in open source ecosystems, has been on the top of the most popular languages on GitHub since 2016[1). Also, JavaScript has been supposed to be the holy grail of cross-platform languages like developing server, mobile, and desktop applications beyond just websites.

While various and fragmented technologies arise and code bases written in JavaScript are larger, the quality cost for debugging and managing code bases is dramatically increasing. The fact that JavaScript does not have grumbling compilers instantly checking code problems makes this worse. There is a research that TypeScript or Flow which supports type checking in JavaScript can prevent 15 percent of the bugs[2).

Static analysis tools come into play at this stage. A research says the relative cost to repair defects at post-product release is 6 times more than at the coding/unit test stage[3). Static analysis tools have significantly reduced the quality cost for the languages like C, C++, and Java by catching code defects at the earlier stage of the development cycle.

Here is a question for you – Why not apply static analysis tools for JavaScript?

Once you have static analysis tools for JavaScript, the tools will help solve the troubles of JavaScript. Although JavaScript is known to be difficult to apply static analysis due to its weak type system and dynamic behavior, fortunately, static analysis tools emerging these days are overcoming such issues.

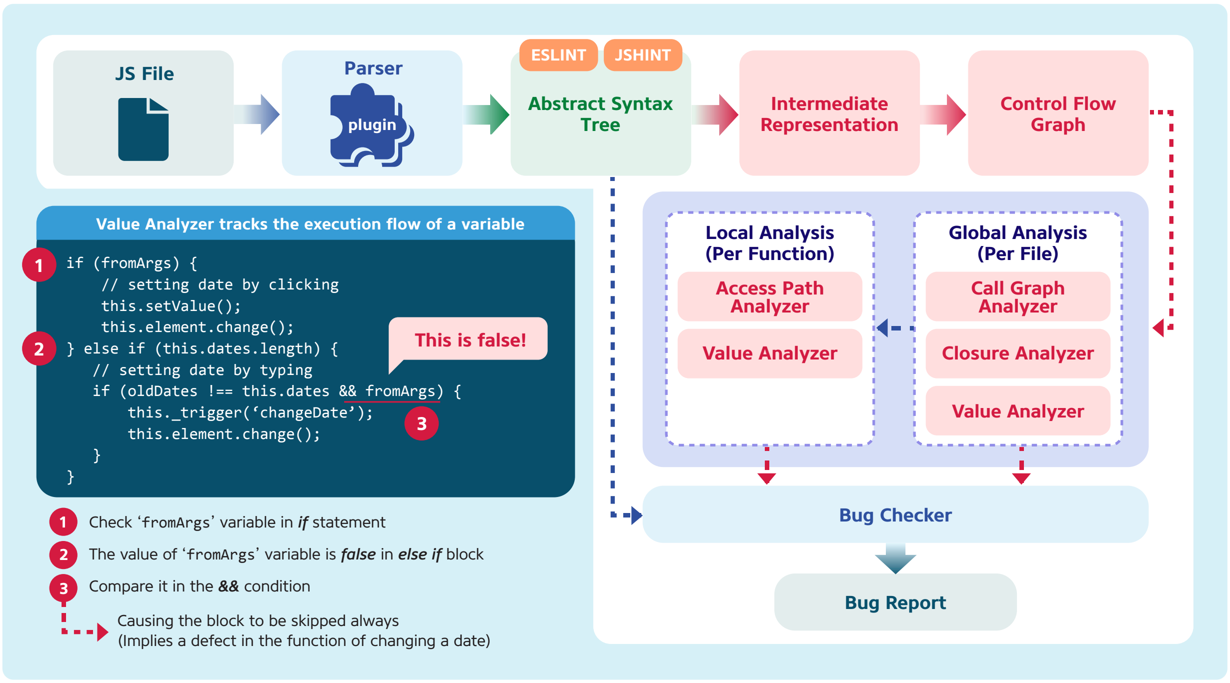
This report describes some of the ways how static analysis tools can help JavaScript folks. Section 2 describes how static analysis tools work. Section 3 shows examples of errors and poor code quality (aka code smells) found by the tool. In its conclusion, this report provides some recommendations for choosing a static analysis tool.

1) "The State of the Octoverse 2017", 2017, (GitHub link: https://octoverse.github.com/)
2) "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript", Zheng Gao, May 2017, (http://ttendency.cs.ucl.ac.uk/projects/type_study/documents/type_study.pdf)
3) "The Economic Impacts of Inadequate Infrastructure for Software Testing", 2002, NIST (https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf)

2. How Static Analysis Works

This section first explains the necessary notions to understand static analysis tools and how they come to play together.

Static analysis tools work much like compilers. Like a compiler, they parse the source code to generate the program's abstract syntax tree (AST) and a symbol table. They convert the AST to an intermediate representation (IR) and construct the control-flow graph (CFG) from the IR.



```
1 if (fromArgs) {
      // setting date by clicking
      this.setValue();
      this.element.change();
2 } else if (this.dates.length) {
     // setting date by typing
     if (oldDates !== this.dates && fromArgs) {
         this._trigger('changeDate');
         this.element.change();
     }
  }
```

1 Check 'fromArgs' variable in *if* statement

2 The value of 'fromArgs' variable is ***false*** in ***else if*** block

3 Compare it in the **&&** condition

Causing the block to be skipped always
(Implies a defect in the function of changing a date)

Figure 1 - An Architecture of Static Analysis Tool

JavaScript Code	Intermediate Representation

`function foo(x) {`
`  var type = "undefind";`
`  if (typeof x === type) {`
`     type =  = "number";`
`  }`
	`[1] assert(typeof x<>1 === type<>3)`
	
	`typeof x<>1              -> Str`
	`x<>1                     -> *`
	`type<>3                  -> "undefind"`
	`typeof x<>1 === type<>3  -> false`

Figure 2 - An Example of IR (Intermediate Representation)

Static analysis tools leverage the AST and CFG to detect specific properties or questionable coding patterns. Simple analysis tools known as linters (such as JSHint or ESLint) constructs only the AST and try to find syntactic and stylistic problems by matching the pattern on AST (e.g., the use of ‘with’ statement).

On the other hand, static analysis tools construct up to the CFG in addition to the AST and look for problems along the execution flow of the whole program (aka, data-flow analysis). They follow the abstract state of the program like the current value of a variable or possible condition of a conditional flow, consequently searching for problems like NULL pointer dereference or invalid function call among modules.

3. JavaScript Errors with Static Analysis Tool

The main discussion here is to show different types of code defects what static analysis tools can prevent.

Recently, Rollbar (a service that provides real-time error monitoring for web application) announced top 10 JavaScript errors they've collected[4]. These run-time errors were caught in the production websites. If static analysis tools can detect these errors earlier in the development stage, the tools can reduce the quality cost and improve the user experience.

[Figure 3] is an example of RangeError thrown when an out-of-range value is passed to a function. For example, 'Number.toFixed()' accepts an argument from 0 to 20, so 'Number.toFixed(25)' in the below throws a RangeError.

RangeError

```
var num = 2.555555;
document.writeln(num.toExponential(-2)); //range error!

num = 2.9999;
document.writeln(num.toFixed(25));  //range error!

num = 2.3456;
document.writeln(num.toPrecision(22));  //range error!
```

Figure 3 - An Example of RangeError (from Rollbar)

These errors can be detected by static analysis tools and linters by querying the AST whether the function to be called is 'toFixed' and its argument. But when the argument is a variable than a constant, only static analysis tools can do the trick because they keep track of the state of a variable.

> *Related with the Common Weakness Enumeration (CWE) - CWE-628 "Function Call with Incorrectly Specified Arguments" is about incorrect arguments which might lead to incorrect behavior and resultant weaknesses.*

4) "Top 10 JavaScript errors from 1000+ projects (and how to avoid them)", 2017, Rollbar (https://rollbar.com/blog/top-10-javascript-errors/)

Another error to be shown is a TypeError caused by referencing a NULL object.

TypeError

```
1 var testArray = ["Test"];
2
3 function testFunction(testArray) {
4   for (var i = 0; i < testArray.length; i++) {
5     console.log(testArray[i]);
6   }
7 }
8
9 testFunction();
```

TypeError

```
1 var test = undefined;
2 test.value = 0;
```

Figure 4 - An Example of TypeError (from Rollbar)

On line 9 of [Figure 4], 'testFunction()' is called without any argument. The 'testArray' argument has an undefined value, so a TypeError is thrown when accessing its 'length' property in the loop.

Static analysis tools can solve this problem. They know the semantics of a function and the call site of it. When the argument is missing and its property is accessed in the function, static analysis tools can ring an alarm with the cause (missing argument at line 9) and highlight the error point (NULL object accessed at line 4).

The example on the right column of [Figure 4] is a nutshell pattern of NULL pointer. As of this, also, static analysis tools recognize the variable 'test' has an undefined value originated from the assignment at line 1. So they can detect a NULL pointer dereference problem at line 2.

> *Related with the CWE - CWE-476 "NULL Pointer Dereference" is about when the application dereferences a pointer that is NULL, typically causing a crash.*

[Figure 5] shows another pattern of null pointer.

TypeError

```
function loadRecommendationsSuccess(data) {
  if (!data) {
      console.warn('error while loading default config values');
  }
  this._saveRecommendedValues(data);
  var configObject = data.resources[0].configurations;
```

Figure 5 - An Example of Insufficient null Check (from Apache Ambari)

On line 2, the argument ‘data’ is checked for NULL. But execution just doesn’t stop, so an undefined value is saved and, finally, a TypeError is thrown when accessing ‘data’ at line 6.

Static analysis tools can help out by detecting whether a variable is consistently checked for NULL in all its usage.

Now, let’s see our survey gathered from analyzing thousands of public JavaScript and TypeScript projects on GitHub regarding run-time errors and code quality.

Here are the top 10 defects we’ve found:

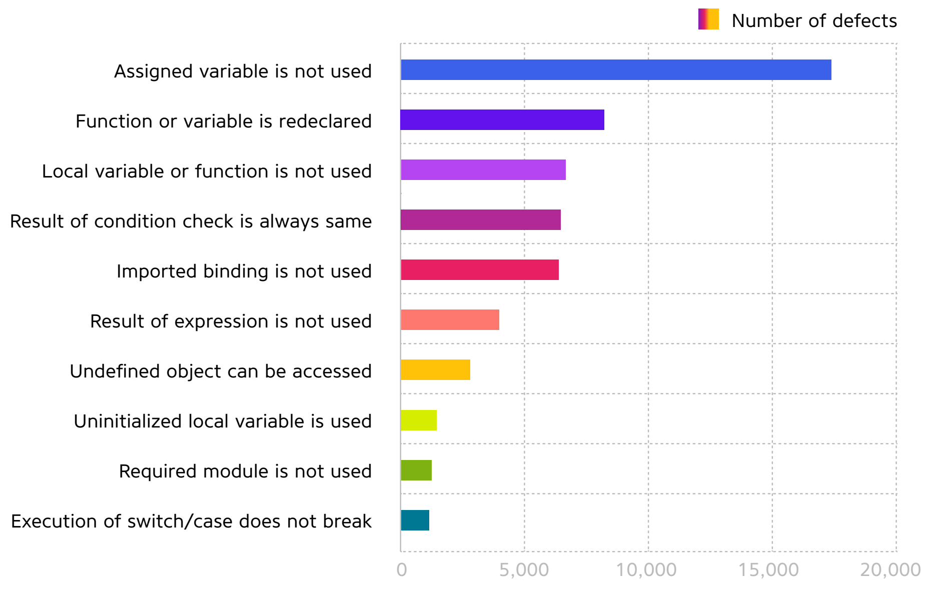


Figure 6 - Number of Defects Detected

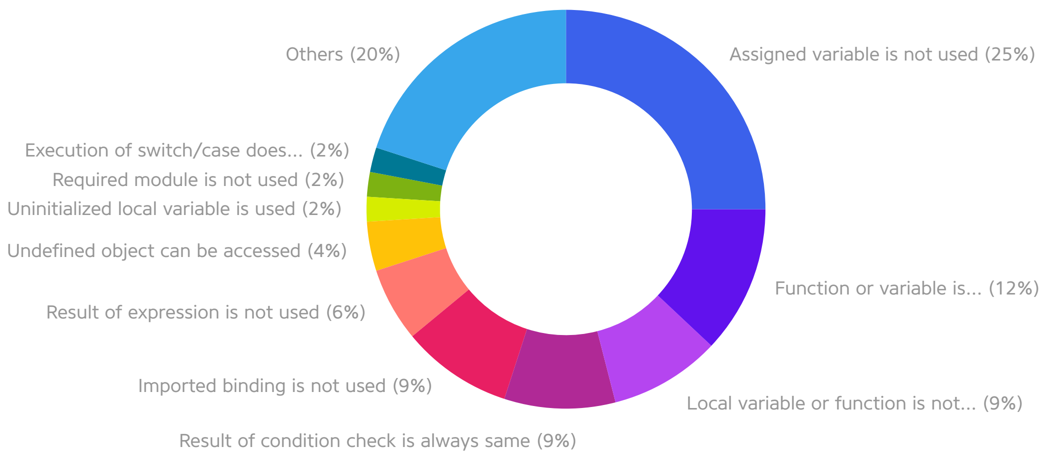


Figure 7 - Defect Types Detected

Many of those are related with unusables like unused or uninitialized local variables and NULL pointer.

Let's dig deeper into each case.

Unused Variables

Declared but unused variables are usually overlooked by developers, but these are not good for future maintenance of code bases (even to the author himself).

This seemingly "trivial" defect can sometimes cause undesired consequences. Below shows an assignment (at line 3) ignored by the subsequent assignment (at line 4), causing the desired parameter 'userName' is not appended.

Value is assigned but ignored

```
1 function getUrl(url) {
2   var target;
3   target = url + "userName" + userName;
4   target = url.replace(/\/|\=|\:|\s/gi,"");
5   return target;
6 }
```

Figure 8 - Example of Unused Variables

Duplicated Variables or Functions

As of duplicate functions, it might imply the developer does not know JavaScript language and/or its new features well, especially when he or she is accustomed to traditional languages. (JavaScript does not support function overloading.)

Although this code defect does not cause an error, it's still in line with the best practices to remove duplications. Duplicated variables or functions increase a technical debt to the development team by confusing which ones are actually used in the code.

Uninitialized Variables

The use of uninitialized variables is probably due to the lack of understanding for the 'var' variable scope and the function hoisting.

In the below, the 'html5' variables are declared at line 2 and 5. Two variables are distinct because 'var' variable has a function scope, but only the variable at line 5 is initialized and the return value coming from line 2 has an undefined value. This would cause an unexpected behavior to the caller.

Use of uninitialized variable

```
define(['isSVG'], function(isSVG) {
  var html5;
  if (!isSVG) {
    ;(function(window, document) {
      var html5 = {
      };

      window.html5 = html5;
    }(typeof window !== 'undefined' ? window : this, document));
   }
   return html5;
});
```

Figure 9 - An Example of Uninitialized Variables

Incautious NULL Checking

NULL checking implies that a TypeError can occur. This is much probable especially in exceptional use cases (the else branch of the code). When unit tests don't come with, developers usually test only for the main use cases (the if branch of the code) or expected argument values. This incurs risks to the user experiences imposing much various use cases.

In the below, the variable 'match' becomes NULL in the else branch at line 9. When it comes at line 12, it is dereferenced and a TypeError throws. This would almost certainly have very undesired consequences for the user experience.

Access null object in else branch

```
1 function _getMarkerAtDocumentPos(editor, pos, preferParent, markCache) {
2   var marks, match;
3
4   ...
5   if (marks.length > 1) {
6     match = marks[marks.length - 2];
7   } else {
8     // We must be outside the root, so there's no containing tag.
9     match = null;
10  }
11
12  return match.mark;
13 }
```

Figure 10 - An Example of Incautious NULL Checking

Also notable is the fact that above examples and statistics are from the code committed to the repository after the final reviews.

Unlike the human developers affected by their emotions and conditions, static analysis tools can find problems in consistent and automatic manner with faster speed. Automated tools can analyze above 130,000 lines of code within fewer than 30 minutes[5].

Also, the tools supporting data-flow analysis can provide an accurate point of the cause for a defect, which can help developers make correction quickly and easily. For example in [Figure 10], the tool can suggest a NULL pointer and its cause in a straightforward manner like:

> *Variable 'match' has a null value originated from assignment 'match = null' at line 9. But its property is accessed at this point.*

To further help your understanding let's take a statistic about "time to fix".

It was calculated as an average from when the defect is detected first to when it's eliminated by the fix (correcting, removing, or triaging).

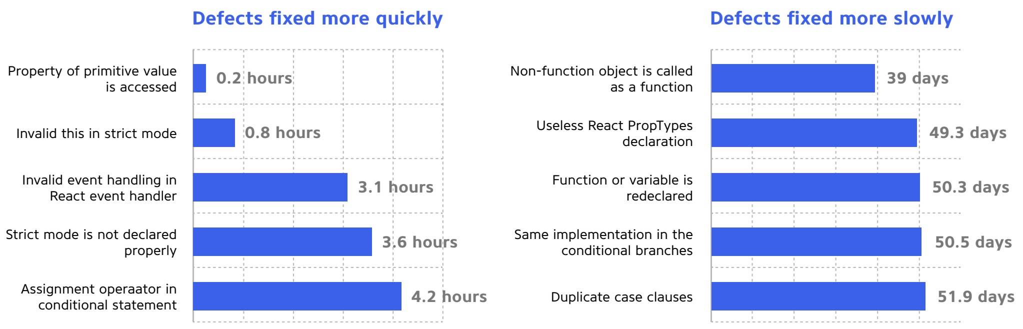


Figure 11 - Average Time to Fix Defects

Defects regarding unusables or duplications (like unused PropTypes, redeclared 'var' variables, the same implementation in if/else branches) seem to be relatively less serious to the developers.

5) "Give Your Defects Some Static Using Automated Static Analyzers To Debug Your Code", Gregory M. Pope, William Oliver, June 2008, Better Software Magazine (https://www.stickyminds.com/better-software-magazine/give-your-defects-some-static)

But some obvious run-time errors or mistakes contrary to the intent of the developer seem to be relatively more serious. Developers are willing to fix the defects such as NULL pointer, incautious NULL checking, misuse of strict mode, incorrect event handling in React (React has its own event system rather than DOM), and assignment operator in conditional statement[6)].

From this finding, we know static analysis tools cannot help and/or educate developers until the tools can find defects implying meaningful impacts in run-time and code quality. Only static analysis tools which understand a data flow of the whole JavaScript program following actual executions will help organizations to improve the development team's productivity and satisfy the return on investment.

6) "This is why code reviews are a good thing", (https://www.reddit.com/r/ProgrammerHumor/comments/4x26u3/this_is_why_code_reviews_are_a_good_thing/)

4. Conclusion

This report shows that static analysis tools can help detect JavaScript code defects effectively and efficiently. In particular:

- Inconsistent NULL checking: Implies run-time errors, especially for exceptional cases, or incautious coding of the author.
- Unusables and/or duplications: Implies technical debt with maintainability.

Static analysis tools engineered to understand the whole JavaScript program can:

- Detect code defects that linters cannot.
- Prevent run-time errors and poor code quality at the earlier stage of the development cycle.
- Help JavaScript developers and testers upgrade their language skills by directly educating them about questionable coding practices.
- Detect code defects human developers might have missed. Note that those examples from Section 3 are about the code committed to the repository after the final reviews. Peer review is very useful but should be supplemented by an automated tool with consistency and faster speed because a human cannot easily get an understanding of the whole program.

To further help your adoption of JavaScript static analysis tools, the following checklist provides a few of recommendations:

Data-flow analysis

It should be aware of data flows in the module and across modules. As the code bases grow larger and modules are getting much more divided, this analysis capability becomes more important.

Lower false alarm rate

Once developers feel the defects found are false positives (claiming the code is a defect when it is actually not), they will not adopt it. So it should maintain low false alarm rate while helping to prevent true problems.

Up-to-dateness

The JavaScript ecosystems change very fast and actively accommodate the changes. The tool needs to support the latest language specifications as well as a wide range of technologies being used such as TypeScript, Flow, JSX, React, and Vue.js.

Up-to-dateness

The JavaScript ecosystems change very fast and actively accommodate the changes. The tool needs to support the latest language specifications as well as a wide range of technologies being used such as TypeScript, Flow, JSX, React, and Vue.js.

Integration with the development workflow

It can be recognized as a tool to blame the developers themselves. Also in the phase where the amount of the development is considerable, it can burden them with the large amount of detection to be corrected and also the wariness of the regression. Therefore, it should provide some of the ways in which they can apply in their daily development, e.g., editor plug-ins or command line tools.

In conclusion, static analysis tools for JavaScript ensure reliability for the release of web applications. It can be incorporated into a development process, a test procedure, or release criteria. As the adoption of JavaScript has proliferated, its static analysis tools will much help organizations (development and project management team) improve their productivity and save the quality cost.

About DeepScan

DeepScan is a leading static analysis tool for JavaScript code. By its advanced data-flow analysis and precise rule sets, DeepScan helps you improve your JavaScript development and releases more reliably – whether you want to write more robust and clean code, better manage the code quality of your software, or decrease quality cost.

Learn about DeepScan at deepscan.io or support@deepscan.io.

kh5325.kim@samsung.com

Kim Kangho (Principal Engineer)

Kangho is product manager for DeepScan,
was the leader of Tizen Web-based SDK.

Strong interest in JavaScript & user relations.

DeepScan