TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript
-
Seminar Report

Nico Fechtner

Technical University of Munich
Department of Informatics
Chair for IT Security
Boltzmannstrae 3, 85748 Garching, Germany
nico.fechtner@tum.de

Table of Contents

Abstract. JavaScript is dynamically and weakly typed which makes it possible to write type inconsistent and therefore buggy code. TypeDevil addresses this issue with a mostly dynamic type inconsistency analysis which is able to effectively warn developers about critical type related bugs. An alternative solution would be to use an additional static type system like e.g. TypeScript. My evaluation shows that both approaches perform quite similar in finding type inconsistencies (?).

1 Introduction

This report is part of the seminar "Common Security Flaws in JavaScript based Applications", which was organized by Paul Muntean from the Chair of IT Security at the Faculty of Informatics of the Technical University of Munich. The seminar took place in the summer semester 2018 and dealt with multiple scientific papers related to JavaScript Security.
I personally took a deeper look at the paper "TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript", published in 2014 by Michael Pradel, Parker Schuh and Koushik Sen. In this report I would like to explain the general problem the paper tries to address, introduce TypeDevil as a possible solution and compare TypeDevil to an alternative approach, namely additional static type systems.

2 Inconsistent Types as the Root Cause of Many Bugs

First of all, I would like to introduce the general problem, TypeDevil wants to solve.
JavaScript has two interesting characteristics we will focus on. On the one hand, JavaScript is dynamically typed. This basically means, that we do not provide any static type annotations to our source code, like you would in statically typed languages, such as e.g. Java, and that types can change during runtime. On the other hand, JavaScript is also weakly typed and thereby very permissive. In order to prevent runtime exceptions, JavaScript performs a lot of automatic type conversions, also known as implicit coercions. Consider listing 1.1. One could expect the output `"Cat Dog Rabbit "`. The actual output however is `"undefinedCat Dog Rabbit "`. In the first iteration of the for-loop, we try to concatenate a string to the value of `outputString`, which is undefined. JavaScript now performs a implicit coercion and converts the value `undefined` to the string `"undefined"`.

```
1 var pets = ["Cat", "Dog", "Rabbit"];
2
3 var outputString;
4
5 for (var i in pets) {
6     outputString += pets[i] + " ";
7 }
8
9 console.log(outputString);
```

Listing 1.1. Implicit Coercions

As we saw, dynamic languages do not require programmers to annotate their programs with type information or to follow any strict typing discipline. This freedom allows programmers to write concise code in short time. However, most code does follow implicit type rules, e.g. only a single type per variable or object property or fixed function signatures. The authors of TypeDevil also state that many bugs are actually violations of these rules. So the freedom offered by dynamic languages often comes at the cost of hidden bugs. Since the language does not enforce any typing discipline, no compile-time warnings are reported if a program uses and combines types inconsistently. Although the code example does not seem that harmful, when you consider critical applications like authentication or payment libraries, of course even a little type error could do some real damage.
TypeDevil wants to address this problem by gathering type observations during run-time, summarizing them into a type graph and finally reporting a set of agressivly filtered type inconsistency warnings. The challenges are on the one hand the absence of static type information and on the other hand the fact that a lot of JavaScript code is polymorphic on purpose.

3 Static and Dynamic Analyses

For better understanding of the following I would like to clearify the difference between static and dynamic analyses first.
The key difference is that within a static analysis you do not have any runtime information, since the analysis runs ahead of time and the only input to the analysis is the sourcecode itself. Therefore it is usually quite hard to predict the exact program behavior. It turns out, that it is even harder to get comprehensive and meaningful results for dynamic languages like JavaScript, because of constructs like the `eval()` function, which interprets the string passed to it at runtime as normal JavaScript code. However the good part about static analysis is that it usually covers all code paths and runs quite fast. Typical examples for static analyses in the context of JavaScript are linters, like ESLint, JSLint or Prettier, which can "detect common programming errors and ensure that certain best practices are followed".
Dynamic analyses are typically characterized through the fact that they do have access to runtime information. Often instrumentation code gets added to the

original source code to keep track of certain characteristics of the program the analysis cares about. After that, this instrumented version of the source code gets executed, the analysis gathers runtime information, e.g. user input, and the results of the analysis get computed. With this approach you can catch certain errors, static analysis wouldn't be able to find. For example, consider again the `eval()` function. With a dynamic analysis you can actually track the code which gets interpreted on the fly. However, the biggest downside of a purely dynamic analysis is, that it naturally doesn't cover all code paths. TypeDevil, as we will see in the following, uses a mostly dynamic approach to gather type observations at runtime.

4 TypeDevils Approach

This section will introduce you to the mechanisms TypeDevil uses to effectively identify and report inconsistent types. In general, there are three main steps, TypeDevil performs: First, instrumentation code gets inserted to the original source code to gather type observations at nearly any point in the program execution. Out of this - for non trivial really huge - set of type observations, TypeDevil builds a TypeGraph, which makes it possible to relatively easily merge type which are structurally equivalent. On the basis of this condensed type graph, TypeDevil identifies inconsistent types and applies a set of merging and pruning techniques to aggressively reduce the number of reported warnings.

4.1 Running Example

Consider listing 1.2, which shows the running example I will use to introduce you to the various steps of TypeDevils pipeline. Note that this is also the example, the authors of TypeDevil used throughout their paper. As you can see, there is a simple function `addWrapped` which takes two arguments and returns the addition of the `v` properties of both arguments if you pass two parameters and returns just the `v` property of the first argument if only one is passed at the function call. Then there is a simple constructor function `Wrapper` which just creates an object with only one property `v`. And lastly `addWrapped` gets called three times with different parameters. Of course it sticks out that the `v` property of the object we pass to `addWrapped` when calling it the third time is a string and not a number. This seems to be a type inconsistency and as we will see, TypeDevil will effectively warn us about this.

```
1  function addWrapped(x, y) {
2      if (y) {
3          return x.v + y.v;
4      } else {
5          return x.v;
6      }
7  }
8
9  function Wrapper(v) {
10     this.v = v;
11 }
12
13 addWrapped({v:23});
14 addWrapped({v:20}, new Wrapper(3));
15 addWrapped({v:"18"}, new Wrapper(5));
```

Listing 1.2. Running Example

4.2 Gathering Type Observations

The first step of TypeDevils pipeline is to gather type observations at runtime. To achieve this, TypeDevil instruments the original source code in two ways. On the one hand, it adds a shadow value to each object and function, containing a unique identifier, which makes it easy to access the objects of functions type name whenever they are referred to. On the other hand, TypeDevil gathers type observations at every relevant code location. A type in the sense of TypeDevil is either a primitive type (boolean, number, string, undefined, null) or a so called record type, which basically maps a named property to a set of types. For record types, TypeDevil differentiates between object types, array types, function types and (function) frame types. A type observation is basically a triple consisting of a basetype, a property and an observed type. In the following, we will look at each instrumentation, TypeDevil adds to the original source of our running example.

Object Literals Consider `addWrapped` gets called for the first time. With the use of the object literal `{v:23}` we create a new object to pass it to the function. First, we append a shadow value holding a unique identifier to the newly created object. A possible unique identifier could just be the string `"object"` concatenated with a unique number, e.g. `"object1"`. After that, we add a type observation for each property of the object. In our example we would add the type observation `(object1, v, number)`, since `"object1"` is the basetype, `v` is the name of the property and `23` is a numeric value.

Property Access TypeDevil also gathers type observations on each property access. In our example, when we call `addWrapped` for the first time, we do not provide a second argument, so we will definitely end up in the else branch on

line 5. There we access the property `v` of the object we passed to the function. So we add the type observation `(object1, v, number)`. As you can see, this is exactly the same type observation we already stored at the creation of the object which, for sure, seems quite redundant. Nevertheless we have to add a type observation for each property access, since code which we do not instrument like third party libraries or code which we just cannot instrument, namely native code, could potentially change the types of properties.

Function Literals Just as with objects, TypeDevil also appends shadow values containing unique identifiers to all function literals. In our example, it would append the unique identifier `"function addWrapped"`, to the function starting at line 1.

Function Calls For each function call, TypeDevil adds two type observations. The first one refers to the receiver of the function and the second one refers to the return type. Consider again the first call of `addWrapped`. TypeDevil adds the type observation `(function addWrapped, this, window)`, since `addWrapped` gets called without an explicit receiver in which case the global object - e.g. `window` - becomes the receiver and `(function addWrapped, return, number)`, since the function returns a numeric value (`23`).

Variable Access We already saw how TypeDevil gathers type observations for each property access. The same thing happens on every local variable access. It is important to note, that similar to the internal implementation of JavaScript itself, TypeDevil treats function arguments as local variables. Like in the previous examples, consider the first call of `addWrapped`. TypeDevil adds the type observation `(frame addWrapped, y, undefined)` when we access the value of `y` which happens to be `undefined`.

4.3 Executing the Program

After instrumenting the original source code, TypeDevil executes the program and dynamically gathers type observation for each construct I mentioned above. The authors of TypeDevil implemented two versions of the analysis. You can either run it on top of node or you can use a custom version of Firefox where they modified Spidermonkey, the JavaScript engine of Firefox, to instrument the JavaScript programs before they get executed. So after the execution we basically have a huge set of type observations we can process further.

4.4 Building the Type Graph and Identifying Inconsistent Types

4.5 Merging and Pruning of Warnings

5 Related Research

First of all, there is a lot of related research on static type analysis. While the first publications in this field restricted JavaScript to a subset which was suitable

for static analysis, [x] showed that it is also possible to perform an effective static type inconsistency analysis while allowing the whole JavaScript language. Of course also for type analyses the typical advantages and disadvantages of static analysis explained in 3 hold true. In the field of dynamic type analysis for JavaScript, TypeDevil is still the state of the art approach, according to [x]. [x] also implemented a combined type analysis which first analyzes programs statically and afterwards refines the results with a subsequent dynamic analysis which implements the basic ideas of TypeDevil. Another research area which tries to solve the general problem of type inconsistencies for dynamic languages is the field of additional type checkers. We will take a closer look at this approach in the following section.

6 Static Type Systems as an Alternative to Dynamic Analysis

Additional static type systems for JavaScript got a lot of attention over the last years, since major tech companies like Google, Facebook or Microsoft are developing them and provide production ready tools for developers. They basically try to solve the same problem like TypeDevil, so they want to provide you with warnings when you are writing type inconsistent code, but instead of TypeDevil you already get these warnings ahead of time. Note that static type systems for dynamic languages also have a lot of additional advantages. For example they enable a mature development experience with rich auto completion and advanced refactoring options. However, in the following we will just focus on type warnings.

6.1 Approach of Additional Static Type Systems

There are a lot of different additional static type systems - also known as static type checkers - for JavaScript. The three most used ones are TypeScript (Microsoft), Flow (Facebook) and the Google Closure Compiler. Basically they all work quite similar. Most of them are a super set of JavaScript, which simply means that every valid JavaScript program is already also for example a TypeScript or a Flow program. This makes the gradual adoption of an additional type system really easy. The approach is to extend the core JavaScript language with additional type annotations, be it in the form of inline types (TypeScript, Flow) or in the form of special comments (Google Closure). Another characteristic of these static type checkers is their mature type inference, which means that you do not have to provide special type annotations at any point in your program per se. Instead, the tools try to infer most of the types for you. For example, when you declare and initialize a variable to a numeric value, the inferred type for that variable would be `number`. If you afterwards try to assign e.g. a string value to this variable, you would get a compiler warning without having to explicitly declare that this variable should only hold numeric values.

6.2 Introduction to TypeScript

To give you the necessary background for the following comparison in 7.2, I would like to introduce you to TypeScript. I choose TypeScript as an example for a static type checker, since it is by far the most used one. Furthermore, [x] showed, that in terms of type inconsistencies, TypeScript and Flow report nearly the exact same warnings. As stated in [x], TypeScript is a syntactic superset of JavaScript. It provides additional syntax for declaring and expressing types, for annotating properties, variables, parameters and return values with types, and for asserting the type of an expression. However, as explained in 6.1, TypeScript also supports rich type inference. For example, consider the example shown in Listing [x]. Without any modification of the source code, you would get the following warning, when compiling this script with the TypeScript compiler: `Operator '+=' cannot be applied to types 'number' and 'number | boolean'`. So TypeScript notes, that the property `a` of the objects which the program iterates over is sometimes a numeric and sometimes a boolean value and prevents you from writing type inconsistent code like this.

```
1 (function() {
2
3     var a = [{a: 23}, {a: 42}, {a: false}];
4     var sum = 0;
5     a.forEach(function(x) {
6         sum += x.a;
7     });
8
9 })();
```

Listing 1.3. inconsistent_foreach.js

7 Evaluation

7.1 Original Results for TypeDevil

7.2 Comparing TypeDevil and TypeScript

Evaluation Setup

Results

7.3 Discussion

In order to permit cross referencing within LNCS-Online, and eventually between different publishers and their online databases, LNCS will, from now on, be standardizing the format of the references. This new feature will increase the visibility of publications and facilitate academic research considerably. Please

base your references on the examples below. References that don't adhere to this style will be reformatted by Springer. You should therefore check your references thoroughly when you receive the final pdf of your paper. The reference section must be complete. You may not omit references. Instructions as to where to find a fuller version of the references are not permissible.

We only accept references written using the latin alphabet. If the title of the book you are referring to is in Russian or Chinese, then please write (in Russian) or (in Chinese) at the end of the transcript or translation of the title.

Please note that proceedings published in LNCS are not cited with their full titles, but with their acronyms!

References

1. Smith, T.F., Waterman, M.S.: Identification of Common Molecular Subsequences. J. Mol. Biol. 147, 195–197 (1981)
2. May, P., Ehrlich, H.C., Steinke, T.: ZIB Structure Prediction Pipeline: Composing a Complex Biological Workflow through Web Services. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 1148–1158. Springer, Heidelberg (2006)
3. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (1999)
4. Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid Information Services for Distributed Resource Sharing. In: 10th IEEE International Symposium on High Performance Distributed Computing, pp. 181–184. IEEE Press, New York (2001)
5. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: an Open Grid Services Architecture for Distributed Systems Integration. Technical report, Global Grid Forum (2002)
6. National Center for Biotechnology Information, `http://www.ncbi.nlm.nih.gov`

Appendix: Springer-Author Discount

LNCS authors are entitled to a 33.3% discount off all Springer publications. Before placing an order, the author should send an email, giving full details of his or her Springer publication, to `orders-HD-individuals@springer.com` to obtain a so-called token. This token is a number, which must be entered when placing an order via the Internet, in order to obtain the discount.

8 Conclusion and Future Work

Here is a checklist of everything the volume editor requires from you:

☐ The final LaTeX source files

☐ A final PDF file

☐ A copyright form, signed by one author on behalf of all of the authors of the paper.

☐ A readme giving the name and email address of the corresponding author.