TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript
-
Seminar Report

Nico Fechtner

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

Table of Contents

Abstract. JavaScripts dynamic and weak type system 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 heavily used in the industry are additional static type system like e.g. TypeScript. A comparing evaluation shows that both approaches perform quite similar in finding type inconsistencies with TypeScript being just slightly better.

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 addresses. JavaScript as a programing language features two striking 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 do in statically typed languages, such as e.g. Java, and that types - for example of local variables - 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 the code example in listing 2.1. One could expect the script to print `"Cat Dog Rabbit "` to the console. 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"` which leads to the unexpected output.

```
var pets = ["Cat", "Dog", "Rabbit"];

var outputString;

for (var i in pets) {
    outputString += pets[i] + " ";
}

console.log(outputString);
```

Listing 2.1: 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 developers 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 further 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 and 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 business sensitive services 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 aggressively 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 and the tool shouldn't report false positives when encountering intended polymorphic constructs.

3 Static and Dynamic Analyses

For better understanding of the following I would like to clarify the difference between static and dynamic analyses first.
The key differentiation point 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 source code 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 precise 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 can not cover all code paths most of the time. TypeDevil 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 phases, TypeDevil goes through: First, instrumentation code gets inserted to the original source code to gather type observations at every relevant point in the program execution. Out of this potentially huge set of type observations, TypeDevil constructs a type graph, which makes it possible to merge types which are structurally equivalent relatively easily. On the basis of the condensed type graph, TypeDevil identifies inconsistent types and applies a set of merging and pruning techniques to drastically reduce the number of reported warnings.

4.1 Running Example

Consider listing 4.1, 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 problem.

```javascript
function addWrapped(x, y) {
    if (y) {
        return x.v + y.v;
    } else {
        return x.v;
    }
}

function Wrapper(v) {
    this.v = v;
}

addWrapped({v:23});
addWrapped({v:20}, new Wrapper(3));
addWrapped({v:"18"}, new Wrapper(5));
```

Listing 4.1: Running Example

4.2 Gathering Type Observations

The first phase of TypeDevil is to gather type observations at runtime. To achieve this, TypeDevil instruments the original source code in two ways. On the one hand, a shadow value gets added 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 saves type observations at every relevant code location to a global set. In the sense of TypeDevil, a type 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 therefore 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 code of our running example.

Object Literals Consider `addWrapped` gets called for the first time. With 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 seems quite redundant at the first glance. 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 we saw on the example of 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. Analogical to that TypeDevil deals with local variable accesses. 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 the value of `y` gets accessed, 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 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. After the instrumented execution TypeDevil holds a huge set of type observations for further processing.

4.4 Building the Type Graph and Identifying Inconsistent Types

Iterating over each type observation, TypeDevil can now produce a type graph sticking to the following heuristic. In the first step, every type becomes a node in the graph. If a type is a record type, there will be an outgoing edge for each property labeled with the name of the property. There is only one node for each occurring primitive type. Figure 1 shows the uncondensed type graph for the running example.
Subsequently, TypeDevil reduces the graph, so that there are no two nodes which are structurally equivalent, which means there are no nodes left which have outgoing edges equally labled and pointing to the same type. Figure 2 shows the condensed type graph for the running example. For example, you can spot that `object1`, `object2`, `object3` and `object5` got merged together, since they are structurally equivalent. Condensing the type graph is extremely important, because the redundancy of the initial graph can really slow down the following operations on the graph.
TypeDevil can now identify inconsistent types. The intuition is again quite simple: Whenever a type has got two or more outgoing edges, labled equally, but pointing to different types, there is a type inconsistency. In our example, the local variables `x` and `y` are obviously of inconsistent types.

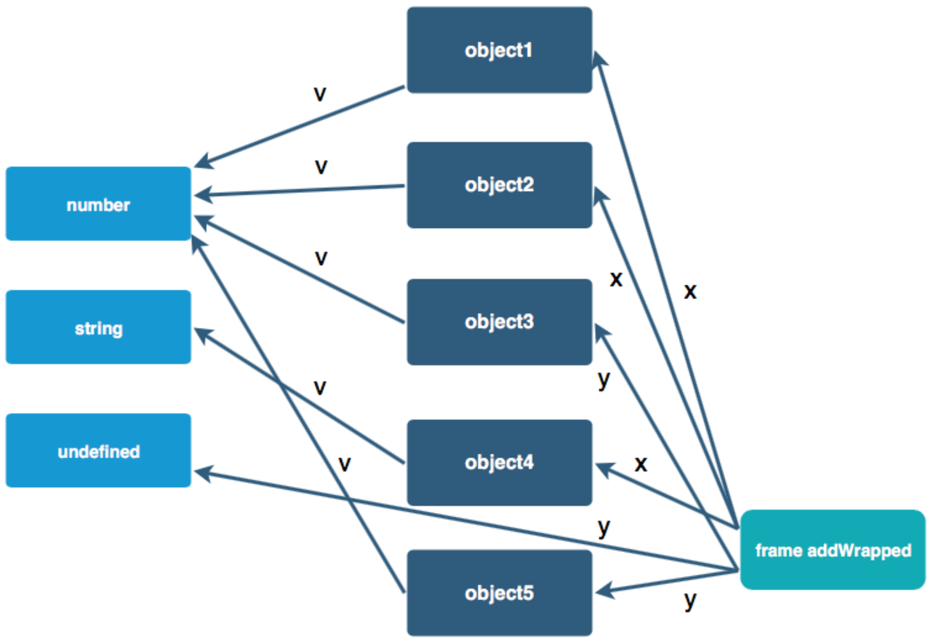

Fig. 1: Uncondensed Type Graph

4.5 Merging and Pruning of Warnings

A naive implementation of the approach explained in 4.4 reports various false positives, since a variable, property, or function may refer to multiple types on purpose. TypeDevil combines two kinds of techniques to distinguish such intended polymorphic code from inconsistency problems. First of all, the authors of TypeDevil developed techniques which try to put warnings which seem to have the same root cause in the same equivalence class. Ensuing, TypeDevil marks warnings which are likely to be likely false positives for pruning. TypeDevil then only keeps one warning per equivalence class if none of the warnings in this class are marked for pruning.
There are a lot of different merging and pruning heuristics which [4] explains in detail. Nevertheless, I want to introduce you to the only pruning technique which utilizes a static analysis which takes place before the main dynamic analysis of TypeDevil, namely pruning by belief analysis.
Consider listing 4.2, which was also used in [4].

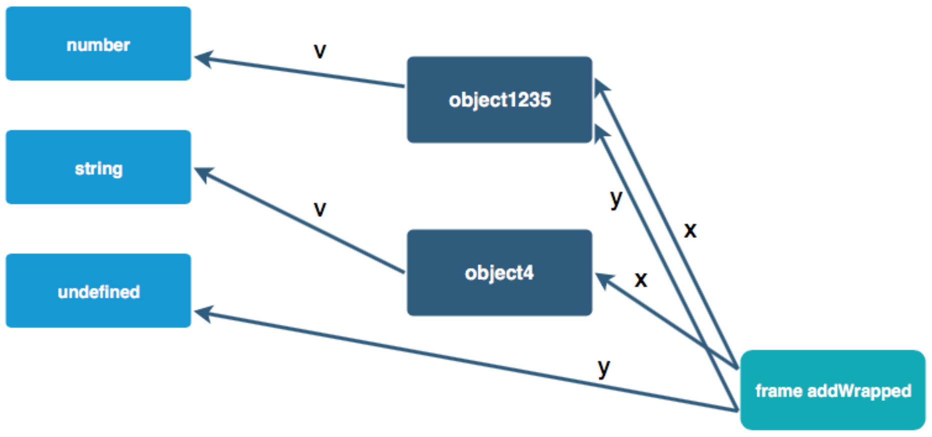

Fig. 2: Condensed Type Graph

```
function BigInteger(a, b, c) {
    this.array = new Array();
    if (a != null) // belief: a may be undefined or null
        if ('number' == typeof a) // belief: a may be number
            this.fromNumber(a, b, c);
        else if (b == null && // belief: b may be undefined
            or null
            'string' != typeof a) // belief: a may be string
            this.fromString(a, 256);
        else
            this.fromString(a, b);
}
```

Listing 4.2: Pruning via Belief Analysis

As you can see, this script uses runtime type checks to distinguish the types that the first two parameters of `BigInteger` may have. This is a really common idiom in industry level JavaScript code. A naive implementation of TypeDevil would report type inconsistencies e.g. when `BigInteger` gets called one time with the first parameter being `null` and the second time with the first parameter being numeric, even though the program is correct. TypeDevil performs a static analysis to extract the programmer's "intent". Common idioms used to check the runtime type of a variable get recognized and the according "belief" gets stored as an string expression at the beginning of the according function. The main analysis then uses this information to prune warnings which are false positives. E.g. for `BigInteger`, TypeDevil would not report a warning if `a` is sometimes `null` and sometimes a `number`.

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, [3] 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.
According to [6], when it comes to dynamic type analysis for JavaScript, TypeDevil is still the state of the art approach. [6] 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 one of additional static type systems - also known as static type checkers. The following section will give deeper insights into this approach.

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 IT companies like Google, Facebook or Microsoft are developing them and provide production ready tools for developers. Trying to solve basically the same problem as TypeDevil, they want to provide developers with warnings for type inconsistent code, but in contrast to dynamic analyses like TypeDevil you already get these warnings ahead of time. Important to note is 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, advanced refactoring options and always up-to-date code documentation. 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 for JavaScript. The three most used ones are TypeScript (Microsoft), Flow (Facebook) and the Google Closure Compiler. Basically all of them behave and work quite similar. Usually, they are a superset of JavaScript, which means that every JavaScript program is also already for example a syntactically valid TypeScript or a Flow program. This makes the gradual adoption of an additional type system really easy. The basic approach is to extend the core JavaScript language with additional type annotations, either in the form of inline types (TypeScript, Flow) or in the form of special comments (Google Closure). Another characteristic of state of the art static type checkers is their mature type inference, which means that developers do not have to provide special type annotations at any point in their program per se. Instead, the tools try to infer most of the types for them.

For example, when declaring and initializing a variable to a numeric value, the inferred type for that variable would be `number`. If there is code which afterwards tries to assign e.g. a `string` value to this variable, there would be a compiler error without having to explicitly declare that this variable should only hold numeric values. Last but not least, keep in mind that these tools are only designed for the development process. They all compile the annotated code back to plain JavaScript which runs in all common JavaScript engines and does not have any runtime type checks.

6.2 Introduction to TypeScript

To provide to necessary background for the upcoming comparison in 7.2, I would like to introduce TypeScript. I choose TypeScript as an example for a static type checker, since it is by far the most used one according to [5]. Furthermore, [2] showed, that in terms of type inconsistencies, TypeScript and Flow report nearly the exact same warnings. As stated in [1], TypeScript is a syntactic superset of JavaScript, which 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 6.1. 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.

```
(function() {
    var a = [{a: 23}, {a: 42}, {a: false}];
    var sum = 0;
    a.forEach(function(x) {
        sum += x.a;
    });
})();
```

Listing 6.1: inconsistent_foreach.js

7 Evaluation

7.1 Original Results for TypeDevil

7.2 Comparing TypeDevil and TypeScript

Evaluation Setup

Results

7.3 Discussion

8 Conclusion and Future Work

References

1. Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard E. Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 257–281. Springer, 2014.
2. Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: quantifying detectable bugs in javascript. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 758–769. IEEE / ACM, 2017.
3. Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In Jens Palsberg and Zhendong Su, editors, *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009.
4. Michael Pradel, Parker Schuh, and Koushik Sen. Typedevil: Dynamic type inconsistency analysis for javascript. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 314–324. IEEE Computer Society, 2015.
5. Raphaël Benitte, Sacha Greif and Michael Rambeau. The state of javascript 2017. `https://stateofjs.com/2017/connections/`, 2017. [Online; accessed 17-June-2018].
6. Tian Huat Tan, Yinxing Xue, Manman Chen, Shuang Liu, Yi Yu, and Jun Sun. Jsfox: integrating static and dynamic type analysis of javascript programs. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 256–258. IEEE Computer Society, 2017.