

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

1	Introduction.....	4
2	Inconsistent Types as the Root Cause of Many Bugs	4
3	Static and Dynamic Analyses.....	5
4	TypeDeviIs Approach.....	6
	4.1 Running Example	6
	4.2 Gathering Type Observations	7
	4.3 Executing the Program	8
	4.4 Building the Type Graph and Identifying Inconsistent Types....	8
	4.5 Merging and Pruning of Warnings	10
5	Related Research	11
6	Static Type Systems as an Alternative to Dynamic Analysis	12
	6.1 Approach of Additional Static Type Systems.....	12
	6.2 Introduction to TypeScript	12
7	Evaluation	14
	7.1 Original Results for TypeDeviI	14
	7.2 Comparing TypeDeviI and TypeScript	14
8	Conclusion and Future Work	18

Abstract. JavaScripts dynamic and weak type system 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 comparative evaluation shows that both approaches perform quite similar in identifying type inconsistencies with TypeScript being about 14% better at finding true positives.

1 Introduction

This report is part of the seminar "Common Security Flaws in JavaScript based Applications" [15], which was held 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 several scientific papers related to JavaScript Security.

I personally took a deeper look at the paper "TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript" [16], published in 2015 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 it to an alternative approach, namely additional static type systems.

2 Inconsistent Types as the Root Cause of Many Bugs

First of all, let me introduce the general problem that TypeDevil tries to solve. JavaScript was developed by Brendan Eich back in 1995 in about ten days and was - as the name suggests - originally just intended for small inline scripts. However, these days there are critical business applications like banking front-ends written entirely in JavaScript. Being one of today's most popular programming languages, JavaScript features two striking characteristics this report will focus on. On the one hand, JavaScript is dynamically typed. This basically means, that developers do not provide any static type annotations to their 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 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, it is being attempted 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.

```

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: Implicit Coercions

As pointed out above, 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. For deeper insights please consider taking a look at the prototype implemented on top of Jalangi [18] by the authors of TypeDevil on GitHub [5].

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 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 which gets passed to `addWrapped` when calling it the third time is a string and not a number. This seems to be a type inconsistency and it will turn out that TypeDevil can effectively warn programmers about this problem.

```

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 2: 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, each instrumentation, TypeDevil adds to the original source code of the running example, will get explained.

Object Literals Consider `addWrapped` gets called for the first time. With the object literal `{v:23}` a new object gets created to pass it to the function. First, a shadow value holding a unique identifier get appended 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, a type observation for each property of the object gets added to the global set of observations. In the example the type observation `(object1, v, number)` would be added, 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 the example, when `addWrapped` gets called for the first time, there

is no second argument provided, so the execution will definitely end up in the else branch on line 5. There the property `v` of the passed object gets accessed. So TypeDevil adds the type observation `(object1, v, number)`. As you can see, this is exactly the same type observation which was already stored at the creation of the object which seems quite redundant at the first glance. Nevertheless TypeDevil has to add a type observation for each property access, since code which does not get instrumented like third party libraries or code which just cannot get instrumented, 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 the 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 Analogical to property accesses, 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 labeled 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, labeled equally, but pointing to different types, there is a type inconsistency. In the running example, the local variables `x` and `y` are obviously of inconsistent types.

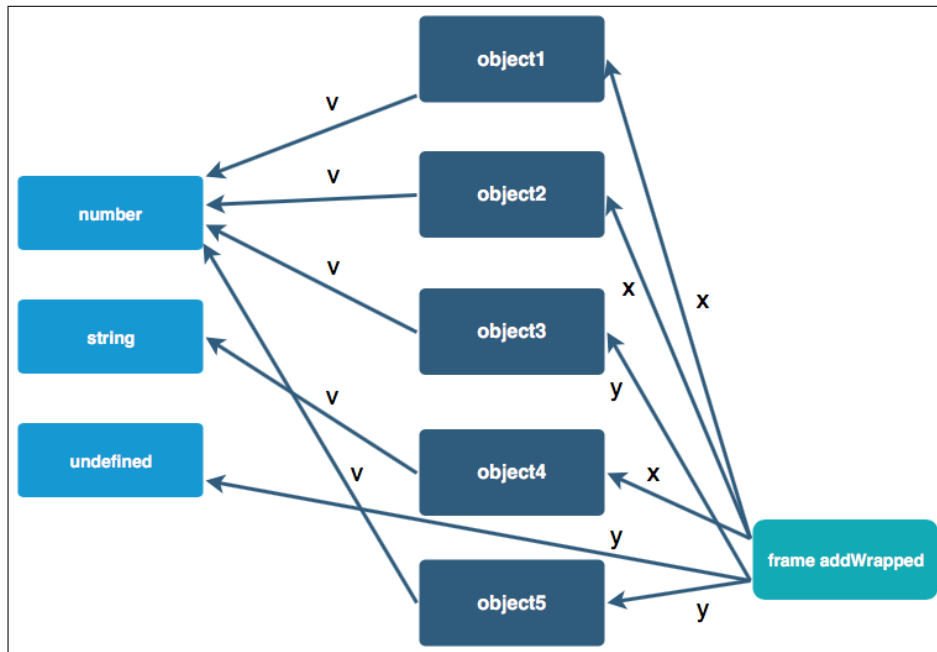


Fig. 1: Uncondensed Type Graph

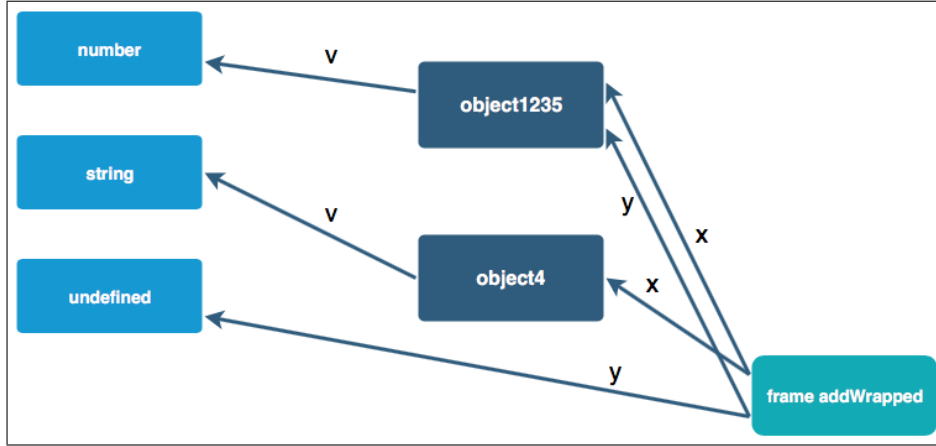


Fig. 2: Condensed Type Graph

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 [16] 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 3, which was also used in [16].

```

1 function BigInteger(a, b, c) {
2   this.array = new Array();
3   if (a !== null) // belief: a may be undefined or null
4     if ('number' === typeof a) // belief: a may be number
5       this.fromNumber(a, b, c);
6   else if (b == null && // belief: b may be undefined
7     or null
8     'string' !== typeof a) // belief: a may be string
9     this.fromString(a, 256);
10  else
11    this.fromString(a, b);

```

Listing 3: 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 a 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, [11] 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 [19], when it comes to dynamic type analysis for JavaScript, TypeDevil is still the state of the art approach. [19] 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. Furthermore, there is a lot of work on other dynamic analyses for JavaScript, often also on top of Jalangi [1], [9], [8], [12]. 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 further explain 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 the focus will be 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 from Microsoft [13], Flow from Facebook [3] and the Google Closure Compiler [10]. 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 common characteristic state of the art static type checkers share 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. Usually, 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, which I choose as an example for a static type checker, since it is by far the most used one according to [17]. Furthermore, [4] showed, that in terms of type inconsistencies, TypeScript and Flow report nearly the exact same warnings.

Microsoft first released TypeScript in 2012 as an open-source programming language to support the development of large enterprise JavaScript applications.

After installing it via npm, the TypeScript compiler can be used to check either plain .js files (with the compiler option `--allowJs`), or annotated .ts files. As stated in [2], 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. Additionally, as explained in 6.1, TypeScript also supports rich type inference. For example, consider the example shown in Listing 4. 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 is aware of the fact, that the property `a` of the objects the `forEach` function iterates over is sometimes a numeric and sometimes a boolean value and prevents you from writing type inconsistent code like this.

```

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

```

Listing 4: Inconsistent ForEach

For situations where type inference alone is not enough, TypeScript allows you to insert type annotations. For example, when executing the script shown in listing 5 without the type annotation `:number`, the second call of `double` would return `NaN`, since it is not possible to multiply a `string` and a `number`. However, with the annotation, TypeScript reports a compiler error, stating that an "argument of type `string` is not assignable to parameter of type `number`". Type annotations are also possible for return types and local variables. To type entire objects, TypeScript offers classes and interfaces which can also be generic since version 2.0.

```

1 function double(x: number) {
2     return x * 2;
3 }
4 double(3);
5 double("hello");

```

Listing 5: Type Annotations

Last but not least, I want to mention a project called DefinitelyTyped [7], in which about 6,500 developers all around the world collaborate to provide type interfaces for nearly 3,600 third-party libraries at the time of writing. This makes it possible to get a strongly typed interface for every major library just by installing the according type definitions via npm.

7 Evaluation

Within this section I would like to summarize the results of the evaluation the authors of TypeDevil performed in 2015 and after that I will compare TypeDevil to TypeScript with the help of an additional evaluation done by me.

7.1 Original Results for TypeDevil

The original evaluation of TypeDevil showed that the tool is able to effectively identify type inconsistencies which lead to incorrect code in real world programs. TypeDevil got evaluated on Sunspider and Octane, which were state of the art performance benchmarks back in 2015 and on seven real world web applications, amongst others Moodle and Joomla. In total, TypeDevil instrumented and analyzed 375,400 lines of code and reported 33 warnings. The authors classified 15 of these warnings as true positives, meaning that the according type inconsistencies really lead to incorrect code. Some of the remaining warnings arose because of bad coding style, e.g. initializing a variable to a `number` but then only using this variable with `array` values, but there are also a few false positives which could not be filtered out. The evaluation further results that condensing the type graph reduces the number of type nodes by 33% on average which is crucial for further operations on the graph. However, even more important seems to be that the merging and pruning techniques reduce 578 warnings to 33 warnings in total. Without this effectiveness, the tool would be nearly impractical in my opinion, since developers would just be overwhelmed by the huge amount of warnings out of which just about 2,6% are true positives. When looking at each technique in separation it turns out that the most impactful heuristic is the pruning of `null` related warnings. The authors developed this technique, since in JavaScript `null` only appears when a developer explicitly uses it to indicate that e.g. a variable can hold `null` or another value.

7.2 Comparing TypeDevil and TypeScript

As an additional contribution I would like to compare TypeDevil to a static type checker. In 6.2, I already explained why I choose TypeScript as an example for an additional static type system. I am also well aware of the fact that TypeDevil is a three year old prototype written by a little group of researchers and TypeScript is a industry approved compiler developed mainly by a major IT company and supported by huge community. In my opinion, a comparison is still justified, since the tools follow completely contrary approaches and TypeDevil is according to [19] still considered the state of the art dynamic solution. In the following, I will point out the evaluation setup I used and present and comment the results I recorded.

Evaluation Setup The 45 scripts I investigated are all little code snippets the authors of TypeDevil used to test TypeDevil itself. Furthermore and in contrast

to the benchmarks and applications, the original evaluation used, they are not dependent on other third party libraries, which makes them really easy to test. You can find the original scripts in TypeDevils GitHub repository [6]. Note that I did not evaluate the scripts which do not contain type inconsistencies, since I am mainly interested in true positives. Each program tests a specific JavaScript construct or a specific filtering technique of TypeDevil. Even though not all of these script are directly taken from real world applications, I think they are nevertheless sufficiently representative since they cover all major JavaScript constructs and most of the scenarios where type inconsistencies can occur. Throughout my evaluation I used version 2.8.1 of the TypeScript Compiler with the compiler options `--noImplicitReturns` and `--strictNullChecks`. These options report errors if a function does not implicitly return in each code path or if `null` or `undefined` get mixed with other types (except `any` and `void`). In the first step of the evaluation I ran all the scripts through TypeDevil, then I compiled the scripts without any modifications with the TypeScript Compiler and in the last step, I annotated the scripts which did not produce errors just by compiling them with type annotations to help the compiler identify the inconsistencies.

Results and Discussion Table 1 shows the number of warnings TypeDevil and TypeScript - with and without annotations - produces. Additionally the most left column shows the number of additional annotations I had to insert to trigger a relevant warning. When it comes to the results, TypeDevil correctly identified 35 of the 45 problematic type inconsistencies. Note that for three scripts, marked in table 1 with an asterisk, TypeDevil was not able to finish the analysis, because the program crashed during execution with the runtime exception `TypeError: Cannot read property 'xyz' of undefined`.

Script	TypeDevil	TypeScript (unannotated)	TypeScript (annotated)	Annotations needed
api	0	1	1	0
callgraph	2	0	1	1
callgraph2	1	0	1	1
constructor	2	0	1	1
constructor2	2	0	1	1
constructor3	2	0	1	1
dataflow_paper	1	0	1	1
duplicate_warnings	1	0	1	1
duplicate_warnings2	1	0	1	1
duplicates	1	0	0	0
eval	0	0	0	0
foreach	1	1	1	0
frame	1	0	1	1
frame2	2	0	1	1
frame3	1	1	1	0
frame4	2	2	2	0
frame5	2	1	1	0
frame6	1	1	1	0
function_args	1	0	0	0
function_args2	1	0	1	1
function_args3	1	0	1	1
global_frame	1	2	2	0
global_frame2	1	2	2	0
global_frame3	1	1	1	0
global_var	0	1	1	0
local_vars_duplicates	2	0	1	1
missing_new	1	0	1	1
missing_new2	1	0	1	1
null_vs_primitive	0	1	1	0
paper	2	1	1	0
polymorphic_property	1	0	1	1
properties	1	0	1	1
r1	0*	1	1	0
r2	0*	0	0	0
r3	0	0	0	0
r4	1	1	1	0
r5	0	1	1	0
r6	0*	0	1	1
this	1	0	1	1
typediff_duplicates	1	2	2	0
typediff_paper	1	0	1	1
typediff	1	0	1	1
typediff2	0	0	1	1
typediff3	1	0	1	1
typediff4	1	0	1	1
Problems detected:	35	16	40	24

Table 1: Results of Comparing TypeDevil to TypeScript

The false negative which surprised me the most was the script with the `eval` function, since this is the type of dynamic behavior one just can not catch with static analyses and you should think that this is where a dynamic analysis like TypeDevil shines. When compiling the scripts without any annotations, the TypeScript compiler identified 16 out of 45 type inconsistencies, which may not seem that much at first glance, but is at least 45% of the errors TypeDevil spots with the advantage that you get these warnings ahead of time without executing the program. In the last phase of my evaluation it looked manually through all scripts for which TypeScript was not able to spot the type inconsistency on its own and added appropriate type annotations to support the compiler. Most of the time it was just as simple as adding a type annotation to a function parameter or specifying the return type of a function. Whenever the type inconsistency was caused by objects which were structurally different, I added a simple class or interface just specifying the properties the object should have and adding types to these properties if needed. After adding these minimal annotations, TypeScript was able to spot 40 out of 45 type inconsistencies ahead of time which makes it - at least for these test scripts - better performing than TypeDevil. Interesting are the scripts where TypeScript was not able to spot the incorrectness. Two expectable false negatives are uses of the `eval` and the `JSON.parse` function, since it is just not possible to predict the behavior and the value to be returned, respectively, with a static analysis. A rather unexpected failure happened in two scripts, where TypeScript was even with the compiler option `--strictNullChecks` not able to detect, that a function parameter was sometimes a `number` (or a `string`, respectively) and sometimes `undefined`. I originally suspected, that TypeScript would be able to catch these inconsistency. For the last script where TypeDevil does not report a type inconsistency, one could argue that this is actually a true negative, since the function, which throws a runtime exception when called without a parameter, gets never called in the script, so technically there is no type inconsistency at all. To sum up, one could say that both tools perform reasonably well at identifying type inconsistencies with TypeScript being just slightly better. However, since TypeDevil is just a prototype, being a developer you would probably always prefer a solid static type checker over TypeDevil. The only scenario I can imagine, where it could be worth it to give TypeDevil a try is when you have a big legacy JavaScript application with hundreds of thousands lines of code where it is not feasible to go through each file manually and add type annotations, since compared to the plain TypeScript compiler, TypeDevil performs more than twice as good. But of course with big applications you also have to consider the 26X to 30X overhead of Jalangi, on which TypeDevil relies, while a static type checker has just a very little overhead ahead of time.

8 Conclusion and Future Work

With this report, I introduced the general problem of JavaScripts dynamic and weak type system enabling type related bugs. I further summarized the key aspects of [16], which proposed TypeDevil as a possible solution for the problem. To look a bit further I additionally compared TypeDevil to TypeScript, a state of the art static type checker, concluding with the fact, the both tools perform quite well with TypeScript being just a little bit better.

Concerning future work I suspect a lot of innovation in the area of static type checkers, since major companies like Microsoft and Facebook are working on those tools. However, I personally hope that someone also picks up the idea of [19] again and actually provides an open-source implementation of an integrated static and dynamic type analysis, since the paper reads really promising, but I was not able to find an actual implementation of this approach.

List of Listing

1	Implicit Coercions	5
2	Running Example	7
3	Pruning via Belief Analysis	11
4	Inconsistent ForEach.....	13
5	Type Annotations	13

List of Figures

1	Uncondensed Type Graph.....	9
2	Condensed Type Graph.....	10

References

1. Thomas H. Austin. *Dynamic Information Flow Analysis for JavaScript in a Web Browser*. PhD thesis, University of California, Santa Cruz, USA, 2013.
2. 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.
3. Facebook. Flow. <https://flow.org/en/>, 2018. [Online; accessed 26-June-2018].
4. 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.
5. GitHub. Typedevil. <https://github.com/Berkeley-Correctness-Group/Jalangi-Berkeley/tree/master/src/js/analyses/inconsistentType>, 2015. [Online; accessed 26-June-2018].
6. GitHub. Typedevil test scripts. <https://github.com/Berkeley-Correctness-Group/Jalangi-Berkeley/tree/master/tests/inconsistentType>, 2015. [Online; accessed 26-June-2018].
7. GitHub. Definitelytyped. <https://github.com/DefinitelyTyped/DefinitelyTyped>, 2018. [Online; accessed 26-June-2018].
8. Liang Gong, Michael Pradel, and Koushik Sen. Jitprof: pinpointing jit-unfriendly javascript code. In Nitto et al. [14], pages 357–368.
9. Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. Dlint: dynamically checking bad coding practices in javascript. In Michal Young and Tao Xie, editors, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 94–105. ACM, 2015.
10. Google. Closure compiler. <https://developers.google.com/closure/compiler/>, 2018. [Online; accessed 26-June-2018].
11. Simon Holm Jensen, Anders Möller, 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.
12. Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. Meminsight: platform-independent memory debugging for javascript. In Nitto et al. [14], pages 345–356.
13. Microsoft. Typescript. <https://www.typescriptlang.org>, 2018. [Online; accessed 26-June-2018].
14. Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. ACM, 2015.
15. Paul Muntean. Common security flaws in javascript based applications. <https://www.sec.in.tum.de/i20/teaching/ss2018/common-security-flaws>, 2018. [Online; accessed 26-June-2018].
16. 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.
17. Raphaël Benitte, Sacha Greif and Michael Rambeau. The state of javascript 2017. <https://stateofjs.com/2017/connections/>, 2017. [Online; accessed 17-June-2018].
 18. Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of javascript. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 615–618. ACM, 2013.
 19. 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.