TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript

Michael Pradel
Parker Schuh
Koushik Sen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-171
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-171.html

October 7, 2014

Acknowledgement

This research is supported in part by NSF Grants CCF-0747390, CCF-1018729, CCF-1423645, and CCF-1018730, and gifts from Mozilla and Samsung.

TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript

Michael Pradel, Parker Schuh, and Koushik Sen
EECS Department
University of California, Berkeley

***Abstract*—Dynamic languages, such as JavaScript, give programmers the freedom to ignore types, and enable them to write concise code in short time. Despite this freedom, many programs follow implicit type rules, for example, that a function has a particular signature or that a property has a particular type. Violations of such implicit type rules often correlate with problems in the program. This paper presents TypeDevil, a mostly dynamic analysis that warns developers about inconsistent types. The key idea is to assign a set of observed types to each variable, property, and function, to merge types based in their structure, and to warn developers about variables, properties, and functions that have inconsistent types. To deal with the pervasiveness of polymorphic behavior in real-world JavaScript programs, we present a set of techniques to remove spurious warnings and to merge related warnings. Applying TypeDevil to widely used benchmark suites and real-world web applications reveals 15 problematic type inconsistencies, including correctness problems, performance problems, and dangerous coding practices.**

I. Introduction

Dynamic languages, such as JavaScript, are becoming increasingly popular for client-side and server-side web applications, traditional desktop applications, and mobile applications. One reason for this popularity is that 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, 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. Even worse, many dynamic languages silently coerce values from one type into another type, leading to incorrect behavior without any obvious sign of misbehavior.

Figure 1 shows three type-related problems that may easily remain unnoticed. Each example is a previously unreported problem that we find in a popular JavaScript benchmark. Figure 1a shows code that concatenates strings into a larger string. Unfortunately, concatenating the initially undefined variable `dnaOutputStr` with a string results in a string that starts with "undefined". Figure 1b shows code that pads a given string with empty characters until it reaches a particular length. Unfortunately, the function returns a `String` object when the given string already has the desired length and a primitive `string` value otherwise, which is problematic because these two types behave differently. Figure 1c shows the constructor of an object and code that modifies the properties of this object. Unfortunately, this modification overwrites properties of type `number` with `undefined`, which causes a crash when running the program with a configuration that slightly differs from the default configuration. Finding these problems is difficult because they do not lead to obvious signs of misbehavior when executing the programs. How can developers detect such problems despite the permissive nature of JavaScript?

All three examples in Figure 1 share the property that a variable, property or function has multiple, *inconsistent types*. In Figure 1a, variable `dnaOutputStr` holds both the `undefined` value and `string` values. In Figure 1b, function `leftPad` sometimes returns an object and sometimes returns a primitive `string` value. In Figure 1c, variable `gb` contains objects that have different structural types.

This paper exploits the observation that inconsistent types often correlate with problems, and presents TypeDevil, a mostly dynamic analysis that detects inconsistent types in JavaScript programs. The analysis associates type information with each variable, property, and function, and records the types of values observed at runtime. Based on this runtime information, the analysis merges types that are structurally equivalent. Finally, the analysis reports potential problems caused by variables, properties, and functions that have multiple inconsistent types. To avoid unnecessary warnings caused by source code that is intentionally polymorphic, the approach aggressively filters and merges warnings, for example, based on "programmer beliefs" [14] inferred from the source code and based on dataflow relations between variables.

Our work is enabled by two observations about JavaScript code that occurs in practice. First, we observe that, even though it is not required by the language, most code follows implicit type rules. We speculate that programmers voluntarily use consistent types because it facilitates humans to reason about code. Second, we observe that inconsistent types either originate from highly polymorphic yet correct code locations, or from code locations where occasional inconsistencies correlate with problems that developers should be aware of. TypeDevil leverages these two observations by inferring types from runtime behavior and by reporting occasionally observed, inconsistent types as likely problems.

Two kinds of existing analyses reason about types in dynamic languages. First, static and dynamic type inference aims at assigning types to variables and functions to show that the program is well-typed [4], [15], [18], [20], [21], [37]. Due to the inherent difficulties of analyzing dynamic languages and

	(a)	(b)	(c)
Program	regexp-dna (SunSpider)	date-format-xparb (SunSpider)	GB Emulator (Octane)

Source code	`var dnaOutputStr;`
	`for (i in seqs) {`
	`  dnaOutputStr += seqs[i].source;`
	`}`
		`String.leftPad = function(val, size, ch) {`
		`  var result = new String(val);`
		`  if (ch == null) ch = " ";`
		`  while (result.length < size) {`
		`  result = ch + result;`
		`  }`
		`  return result;`
		`}`
			`function GameBoyCanvas() {`
			`  this.width = 160;`
			`  this.height = 144;`
			`}`
			`function initNewCanvas() {`
			`  gb.canvas.width = gb.canvas.clientWidth;`
			`  gb.canvas.height = gb.canvas.clientHeight;`
			`}`

Inconsistent types	Variable `dnaOutputStr` has types `string` and `undefined`	Function `leftPad` has return types object and `string`	Variable `gb` has inconsistent object types (e.g., `gb.canvas.width` has types `number` and `undefined`)
Problem	Incorrect string value "undefinedGTAGG.."	Return values behave differently depending on the length of the given string	Crash when changing the emulator settings

Fig. 1. Real-world problems related to inconsistent types.

due to intentionally polymorphic code, this approach either requires type annotations, or it leads to various type errors that include many false positives. TypeDevil has a relatively low number of false positives because it uses a mostly dynamic analysis, because it focuses on individual problematic code locations instead of inferring sound types for the entire program, and because it uses a set of novel techniques to deal with intentionally polymorphic code. Second, optimizing JIT compilers leverage static and dynamic type inference to emit code specialized towards particular runtime types [19], [22], [31]. Instead of improving the performance of existing programs, TypeDevil warns developers about problematic code locations.

To evaluate TypeDevil, we apply it to popular JavaScript benchmarks and to real-world web applications. In total, the analysis reports 33 type inconsistencies, of which at least 15 correspond to problematic code. The detected inconsistencies lead to incorrect but not necessarily crashing behavior, such as Figure 1a, expose error-prone programming practices, such as Figure 1b, and cause crashes and other unintended behavior in executions that are slightly different from the analyzed executions, such as Figure 1c. The most prevalent root cause for inconsistent types is that a variable, property, or function is unintentionally `undefined`. However, simply reporting all occurrences of `undefined` would not only report many false positives but also miss four of the 15 problems that TypeDevil detects.

In summary, this work contributes the following:

- *Inconsistent types as likely problems*. We present the notion of inconsistent types as a pattern of likely problems in programs written in dynamic languages. (Section III)
- *An analysis to find inconsistent types*. We present TypeDevil, a practical, mostly dynamic analysis that detects inconsistent types in JavaScript programs. In contrast to traditional type inference, TypeDevil does not aim at showing the well-typedness of a program, but it focuses on individual variables, properties, and functions that have inconsistent types. (Section IV)
- *Empirical evidence*. We report the results of an extensive evaluation with widely used JavaScript programs, and we show that TypeDevil reveals previously unreported problems in these programs. (Section V)

II. Overview and Example

This section illustrates our approach with a simple JavaScript program. The program in Figure 2a defines a function `addWrapped()` that takes two numbers, each wrapped into an object, and returns the sum of the two numbers. If only one argument is given, then the function returns the number wrapped by this argument. Wrapped numbers are objects with a property `v` that contains the number. To wrap a number, one can either call the `Wrapper()` constructor or use an object literal, such as `{v:23}`. The program calls `addWrapped()` three times, passing a single argument in line 11, and two arguments in each of lines 12 and 14. The intended behavior of the program is that all three calls of `addWrapped()` return `23`. However, the call in line 14 accidentally uses a string `"18"` instead of the number `18`. Since JavaScript silently coerces values from one type to another, the program executes without any warning. Line 3 coerces the number `5` into a string `"5"` and then concatenates it with the string `"18"`. As a result, the call in line 14 returns `"185"`, and not the expected number `23`.

TypeDevil detects such problems by dynamically analyzing the program and by reporting variables, properties, and functions that have inconsistent types. The approach has three steps, which we illustrate in the following.

Gathering Type Observations: The first step is a dynamic analysis that gathers type observations for each variable, property, and function that the analyzed execution refers to. TypeDevil instruments the program by adding code that records the type of each reference. Types in JavaScript are structural types, and TypeDevil represents types as a record of typed properties. For example, the type of the object created in line 11 is a record with a single entry `v` of type `number`. To bound the number of types, the analysis considers all objects allocated at the same code location and all functions defined at the same code location to have a single type.

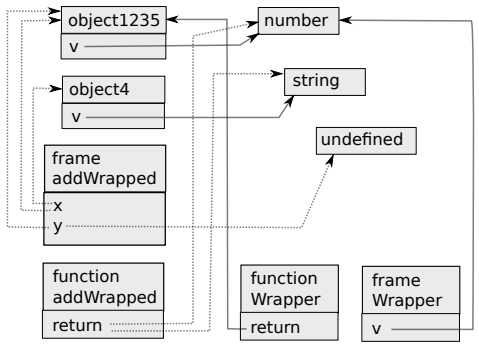
(a) Source code:

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

(b) Type observations (sorted by first appearance):

Base	Prop.	Type	Lines
object1	v	number	11, 5
frame addWrapped	y	undefined	2
frame addWrapped	x	object1	5
function addWrapped	return	number	11, 12
object2	v	number	12, 3
frame Wrapper	v	number	9
object3	v	number	12, 3
function Wrapper	return	object3	12
frame addWrapped	y	object3	2, 3
frame addWrapped	x	object2	3
object4	v	string	13, 3
object5	v	number	13, 3
function Wrapper	return	object5	13
frame addWrapped	y	object5	2, 3
frame addWrapped	x	object4	3
function addWrapped	return	string	13

(c) Condensed type graph:



(d) Reported type inconsistencies:

- Argument `x` of `addWrapped()` has inconsistent object types: `x.v` is `number` (lines 11 and 12) and `string` (line 14).
- `addWrapped()` has inconsistent return types `number` (lines 11 and 12) and `string` (line 14).

Fig. 2. Running example.

Executing the program in Figure 2a yields the type observations summarized in Figure 2b.[1] Each row of the table describes the types observed for a property of a base type. We use the same representation of type observations for types of properties, variables, and functions. For example, the first row describes that the object created at line 11, called "object1", has a property `v` of type `number`. The second row describes that the local variable `y` of `addWrapped()` is observed to have type `undefined`. The fourth row describes that the return type of function `addWrapped()` is observed to be `number`.

Summarizing Observations into a Type Graph: After gathering type observations at runtime, TypeDevil merges all observations into a type graph. Nodes in the graph represent types and labeled edges represent properties. An edge p from type t_1 to type t_2 means that t_1 has a property p of type t_2. Figure 2c shows the type graph for our example. For illustration purposes, we represent the outgoing edges of a node as a record. TypeDevil merges type nodes that are structurally equivalent. For example, object1, object2, object3, and object5 all have a single property `v` of type `number`, and therefore are represented as a single type node object1235.

Reporting Problematic Inconsistencies: The final step of TypeDevil is to report type inconsistencies. Initially, the analysis considers each property of a type as inconsistent where the type node has more than one outgoing edge labeled with this property. In Figure 2c, these property edges are dashed. For example, the property that represents the return value of `addWrapped()` is inconsistent because it points both to the number node and to the string node. To avoid overwhelming developers with spurious and duplicate warnings, TypeDevil aggressively filters and merges inconsistencies. For the example, the analysis filters the inconsistency of the argument `y` of `addWrapped()` because the program expects `y` to be `undefined`, as indicated by the check in line 2. As a result, TypeDevil reports two warnings (Figure 2d), which point the developer to the incorrect argument passed to `addWrapped()` in line 14 and to the resulting incorrect return value.

[1]The table omits some type observations for brevity.

III. Problem Statement

In the following, we define the most important terms used throughout the paper and the problem we are addressing.

Definition 1 (Type)
A type is either a primitive type (boolean, number, string, undefined, or null) or a record type $\mathcal{P} \rightarrow 2^{\mathcal{T}}$ that maps named properties to sets of types. A record type represents one of the following kinds of types:

- *An* object type, *where properties represent object properties.*
- *An* array type, *where properties represent indices.*
- *A* function type, *where the "$this$" and "$return$" properties represent the receiver and the return value, respectively.*
- *A* frame type, *where properties represent local variables of a function.*

Treating objects, arrays, functions, and function call frames as a single concept simplifies the description and implementation of TypeDevil. Each property of a record type maps to a set of types because a property may point to multiple different types. The record of a function type does not include the parameters of the function because parameters are treated as local variables in the function's frame type. Our representation of types ignores the prototype of an object.

Two types are consistent if both types are the same, if both types are structurally equivalent, or if one type is a structural subtype of the other type.

Definition 2 (Consistent Types)
Types t_1 and t_2 are consistent, $t_1 \sim t_2$, if and only if one of the following holds:

- *Both t_1 and t_2 are primitive types and $t_1 = t_2$.*
- *Both t_1 and t_2 are record types and all of the following hold:*
  - *$kind(t_1) = kind(t_2)$, where $kind(t)$ is either object, array, function, or frame.*
  - *The sets of properties of t_1 and t_2 are equal or a subset of each other: $dom(t_1) \subseteq dom(t_2)$ or $dom(t_2) \subseteq dom(t_1)$.*
  - *All common properties of t_1 and t_2 point to types that are consistent with each other:*
    $\forall p \in dom(t_1) \cap dom(t_2)$
    $\forall t'_1 \in t_1.p \,.\, \exists t'_2 \in t_2.p \,.\, t'_1 \sim t'_2$ and
    $\forall t'_2 \in t_2.p \,.\, \exists t'_1 \in t_1.p \,.\, t'_1 \sim t'_2$

We say that two types are *inconsistent* if they are not consistent according to the above definition. The problem addressed in this paper is how to detect inconsistent types that are likely programming errors in JavaScript programs.

IV. Approach

This section describes the TypeDevil approach in detail.

A. Gathering Type Observations

The first step of TypeDevil is to instrument and execute the program to gather information about the types of variables, properties, and functions. The result of this first step is a set of *type observations*. Each observation is a triple $(base, prop, type)$, where $base$ is the name of a type, $prop$ is the name of a property of the $base$ type, and $type$ is the name of the type that $prop$ is observed to refer to.

To gather type observations, TypeDevil instruments the program in two ways. First, we add instrumentation code that creates type observations and that stores them into a global set of observations. Second, we add instrumentation code that attaches a shadow value to each object. This shadow value stores the name of the type we associate with the object, to easily access the object's type name whenever the object is referred to. The dynamic analysis is implemented on top of Jalangi; details on the instrumentation and the implementation of shadow values are available in [36]. The following describes the instrumentation points where TypeDevil adds code.

Object literals: To keep track of the type of newly created objects, TypeDevil adds instrumentation code to every object literal. The instrumentation code creates one type observation for each property of the object literal, where $base$ is "object" concatenated with a unique identifier of the object literal's source code location, $prop$ is the name of the object property, and $type$ is the type name of the object that $prop$ refers to. The instrumented code stores the value of $base$ as the shadow value of the newly created object. For example, the object literal in line 11 of Figure 2 leads to a type observation ("$object1$","v","$number$"). When executing this line, TypeDevil attaches the shadow value "$object1$" to the newly created object.

Get property and put property: To keep track of the types of object properties, TypeDevil adds instrumentation code to every source code location that reads or writes an object's property. The instrumented code creates a type observation where $base$ is the type name of the base object, $prop$ is the name of the property, and $type$ is the type name of the value that is read or written. These type observations may replicate information that is already known. We nevertheless record a type observation at every property access because properties may be modified by uninstrumented code, such as native functions. For example, line 3 of Figure 2 reads property `v` of the base object `x`. When this line is reached for the first time (via the call in line 12), it leads to a type observation ("$object2$","v","$number$").

Function literals: To identify function types, TypeDevil adds instrumentation code to every function literal. The instrumented code attaches to the new function object a shadow value, which is "$function$" concatenated with a unique identifier of the function definition site. The value will be used to identify the function's type whenever this function is referred to. For example, when executing the function declaration statement in line 1 of Figure 2, the instrumented program attaches the type name "$function\ addWrapped$" to the newly created function object.

Function calls: To keep track of function types, TypeDevil adds instrumentation code to every function call. The instrumented code records the call's receiver type and return type by creating type observations where $base$ is the name of the function type, $prop$ is "$this$" and "$return$", respectively, and $type$ is the type of the receiver object `this` and the return value, respectively. If the function has no return value, the program records a type observation where $type$ is "$undefined$". At function call sites, TypeDevil does not record the types of function parameters; instead, these observations are recorded when the callee accesses the arguments. If the function call is a constructor call, TypeDevil also creates type observations for the newly created object, as described above for object literals. For example, when executing the call in line 11 of Figure 2, TypeDevil records the following type observations: ("$function\ addWrapped$", "$this$", "$window$") and ("$function\ addWrapped$", "$return$", "$number$"), where "$window$" refers to the global object, which in JavaScript is assigned to `this` in the absence of another receiver.

Function enter: To enable the analysis to keep track of the type of local variables, TypeDevil instruments each function body by adding code at the function entry. The added code associates the name of the frame type of the function with the current stack frame object. The name of the frame type is "$frame$" concatenated with a unique identifier of the source code location of the function definition. The analysis uses the frame type to keep track of the types of local variables of the function. For example, TypeDevil adds instrumentation code to the entry of the function defined in line 1 of Figure 2. When entering this function, the instrumented program associates the name "$frame\ addWrapped$" with the current stack frame.

Variable reads and writes: To keep track of the type of local variables, including function parameters, TypeDevil adds instrumentation code to every variable read and write. The instrumented program creates a type observation where $base$ is the frame type of the stack frame that defines the variable, $prop$ is the name of the local variable, and $type$ is the type of the value that is read or written. For example, when the call in line 11 of Figure 2 leads to a read of variable `y` in line 2, the instrumented code adds a type observation that records the variable's type: $(\text{"}frame\ addWrapped\text{"}, \text{"}y\text{"}, \text{"}undefined\text{"})$

In JavaScript, a variable can be a local variable of the function f that is currently on top of the call stack, a variable accessible via the closure created when f was defined, or a global variable. TypeDevil assigns each local variable to the frame where it is defined, and it assigns global variables to a special global frame.

B. Summarizing Observations into a Type Graph

Executing a program that is instrumented as described in the previous section yields a set of type observations. The next step of TypeDevil is to summarize these observations into a graph that allows for identifying inconsistent types.

Definition 3 (Type Graph)
The type graph G of an execution of a program is a directed graph $(\mathcal{T}, \mathcal{P})$. The nodes $\mathcal{T}$ represent types observed during the execution. An edge $(t_1, t_2, p) \in \mathcal{P}$ represents that the property p of t_1 has been observed to point to a value of type t_2.

To create an initial type graph, TypeDevil iterates through all type observations. For each observation $(base, prop, type)$, the analysis checks if there are nodes in $\mathcal{T}$ that represent $base$ and $type$, and creates such nodes if they do not yet exist. Then, TypeDevil checks if there exists an edge that is labeled with $prop$ from $base$ to $type$, and adds such an edge to the graph if it does not yet exist.

TypeDevil leverages the type graph to identify inconsistent types by searching for nodes that have multiple outgoing transitions labeled with the same property name. Performing this search on the initial type graph would lead to many spurious warnings because in the initial type graph, different nodes may represent structurally equivalent types. For example, in the initial type graph for Figure 2 (graph not shown), the node that represents $function\ Wrapper$ has two outgoing transitions labeled with $return$, each pointing to an object type that has a single property v of type $number$. Reporting inconsistent types based in the initial type graph would lead to a warning even though all objects passed to `Wrapper()` are structurally equivalent.

To avoid the problem of reporting inconsistencies between structurally equivalent types, we condense the type graph by merging equivalent types.

Definition 4 (Equivalent types)
Two types t_1 and t_2 are equivalent, $t_1 \equiv t_2$, if and only if one of the following holds:

- *Both t_1 and t_2 are primitive types and $t_1 = t_2$.*
- *Both t_1 and t_2 are record types and all of the following hold:*
  - *$kind(t_1) = kind(t_2)$, where $kind(t)$ is either object, array, function, or frame.*
  - *t_1 and t_2 have the same properties and each property points to types that are equivalent: $dom(t_1) = dom(t_2)$ and $\forall p \in dom(t_1)\ .\ t_1.p \equiv t_2.p$.*

TypeDevil merges equivalent type nodes of the type graph to obtain a condensed type graph.

Definition 5 (Condensed Type Graph)
The condensed type graph G of an execution of a program is a type graph, for which there do not exist any two nodes that are equivalent.

Algorithm 1 describes our approach for producing a condensed type graph from the initial type graph. The basic idea is to initially assume that all types that have the same kind and the same set of property names are equivalent, and to repeatedly split types that turn out to be not equivalent when following their outgoing edges. The algorithm initializes and refines a map M that assigns type nodes of the initial type graph to type nodes of the condensed type graph. At first, the helper function $initTypeMap()$ initializes the map M by creating a condensed type node for each set of initial type nodes that have the same kind and the same set of property names.

The main part of the algorithm (lines 3 to 11) repeatedly refines M until each type node is equivalent to the condensed type node it maps to. For this purpose, the algorithm searches for a type node t that is currently mapped to a non-equivalent type node, and assigns it to another type node instead. If there exists another type node t' that is equivalent to t according to Definition 4, then the algorithm maps t to this type node t'.

Algorithm 1 Condense type graph.

Input: Type graph $G = (\mathcal{T}, \mathcal{P})$
Output: Condensed type graph $G' = (\mathcal{T}', \mathcal{P}')$

```
1:  M ← initTypeMap(G)                          ▷ map T → T'
2:  changed ← true
3:  while changed do
4:    changed ← false
5:    for all t ∈ T do
6:      if t ≢ M(t) then
7:        if ∃t' ∈ T with t ≠ t' and t ≡ t' then
8:          M(t) ← t'
9:        else
10:         M(t) ← t
11:       changed ← true
12: T' ← ∅
13: P' ← ∅
14: for all t ∈ range(M) do
15:   T' ← T' ∪ mergeNodes({t' ∈ T with M(t') = t})
16:   P' ← P' ∪ mergeEdges({t' ∈ T with M(t') = t}, M)
17: return (T', P')
```

Otherwise, the algorithm creates a new condensed type node by mapping t to itself.

The final part of the algorithm (lines 12 to 16) creates the condensed type graph $(\mathcal{T}', \mathcal{P}')$ by merging type nodes and property edges according to the map M. The helper function $mergeNodes()$ creates a single type node that combines the type names of all given type nodes. Likewise, the helper function $mergeEdges()$ maps the outgoing transitions of the given set of type nodes to the respective nodes in the condensed type graph.

Figure 2c shows the condensed type graph for our running example.

C. Reporting Inconsistencies that are Likely Problems

The final step of TypeDevil is to report inconsistent types that are likely problems. The basic idea is to report a warning for each type node that has multiple outgoing transitions labeled with the same property name.

Definition 6 (Inconsistent Type Warning)
Given a condensed type graph $G = (\mathcal{T}, \mathcal{P})$, an inconsistent type warning is a triple $(t, p, \mathcal{T}_i)$, where $t \in \mathcal{T}$, p is a property name, $\mathcal{T}_i = \{t_i | (t, t_i, p) \in \mathcal{P}\}$, and $|\mathcal{T}_i| \geq 2$.

For example, the first warning of Figure 2d is $(frame\ addWrapped, x, \{object1235, object4\})$.

A naive implementation of this idea reports various warnings that correspond to harmless code. The reason is that a variable, property, or function may refer to multiple types on purpose. TypeDevil heuristically distinguishes such *intended polymorphism* from inconsistency problems by combining two kinds of techniques. First, we use techniques that assign warnings to equivalence classes so that warnings that have the same root cause are in the same class (Sections IV-C1 to IV-C3). Second, we use techniques that mark warnings that are likely false positives for pruning (Sections IV-C4 to IV-C7). TypeDevil computes the final set of warnings by keeping exactly one warning per equivalence class, if and only if none of the warnings in this equivalence class is marked for pruning.

1) Merging by Dataflow Relations: If a single value is referred from multiple references, TypeDevil may report multiple warnings. For example, consider the code on the right. For this simple example, a naive implementation of TypeDevil gives a total of four warnings that report inconsistent types for the argument of `f()`, the argument of `g()`, the return value of `f()` and the return value of `g()`, respectively.

```
function f(x) { return g(x); }
function g(a) { return a; }
f(23); f({p:"abc"});
```

To avoid such duplicate warnings, the analysis gathers a dynamic call graph and merges warnings that may refer to the same value based in potential dataflow relations. Specifically, TypeDevil merges two warnings $(t_1, p_1, \mathcal{T}_{i1})$ and $(t_2, p_2, \mathcal{T}_{i2})$ if they have the same set of inconsistent types, $\mathcal{T}_{i1} = \mathcal{T}_{i2}$, and if any of the following holds. First, t_1 and t_2 are frame types, and there is a call graph edge from the function of t_1 to the function of t_2. This case addresses duplicate warnings due to a function that passes a local variable as a parameter to another function. Second, t_1 is a frame type, t_2 is a function type, p_2 is "$return$", and t_1 and t_2 refer to the same function. This case addresses duplicate warnings due to a function that returns a local variable. Third, both t_1 and t_2 are function types, both p_1 and p_2 are "$return$", and there exists a call graph edge between the function of t_1 and the function of t_2. This case addresses duplicate warnings due to a function that calls another function and returns the other function's return value. Applying this approach to the above example merges all warnings into a single warning.

(a) Source code:

```
var x, y;
x = {a:5};
y = {myX:x, b:true};
x = {a:"hi there"};
y = {myX:x, b:true};
```

(b) Condensed type graph:

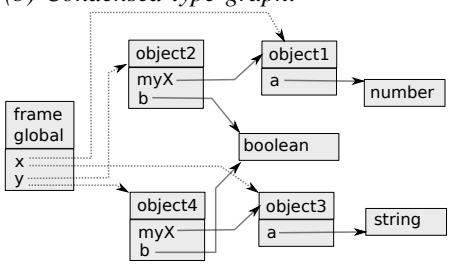


Fig. 3. Example for merging by type diff.

2) Merging by Type Diff: A reference with inconsistent types may cause other references to have inconsistent types because the latter indirectly refers to the first. For example, consider the program in Figure 3a and its condensed type graph in Figure 3b. Variable `x` holds values of inconsistent types because `x.a` may refer to both `number` and `string`. Since `y` refers to the objects assigned to `x` via the property `myX`, variable `y` also contains values of inconsistent types. A naive implementation of TypeDevil reports two warnings, one for `x` and one for `y`.

To merge such warnings, we compute a concise representation of the differences between inconsistent types:

Definition 7 (Type Diff)
Given a condensed type graph G, the type diff $\mathcal{D}$ of a warning $(t, p, \mathcal{T}_i)$ is the maximum set of pairs $(Q, \mathcal{T}_d)$ where

- *Q is a sequence of property names where $target(t_1, Q)$ and $target(t_2, Q)$ are different primitive types or have different kinds for some $t_1, t_2 \in \mathcal{T}_i$ ($target(t, Q)$ is the node reached when following Q from t).*
- *$\mathcal{T}_d$ is the set of type nodes reached when following the path Q from t_1 and t_2*

For Figure 3, the type diff of the inconsistency warning for `x` is $\{([\text{"}a\text{"}], \{\text{"}number\text{"}, \text{"}string\text{"}\})\}$, meaning that `x.a` has types `number` and `string`. The type diff of the inconsistency warning for `y` is $\{([\text{"}myX\text{"}, \text{"}a\text{"}], \{\text{"}number\text{"}, \text{"}string\text{"}\})\}$, meaning that `y.myX.a` has types `number` and `string`.

TypeDevil leverages the type diff to merge related warnings. Intuitively, we merge two warnings if the type diff of one warning is included in the type diff of the other warning. More formally, TypeDevil merges two warnings with type diffs $\mathcal{D}_1$ and $\mathcal{D}_2$ if there exists a bijective mapping between the pairs from $\mathcal{D}_1$ and $\mathcal{D}_2$ so that for each mapped pair $(Q_1, \mathcal{T}_{d1})$ and

```
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);
}
```

Fig. 4. Example for belief analysis (code from Octane's crypto benchmark).

TABLE I
CODE IDIOMS CONSIDERED BY THE STATIC BELIEF ANALYSIS.

Code idiom (`x` is a local variable)	Inferred belief(s)
Expression `typeof x === 'Y'` or `typeof x !== 'Y'` inside a conditional	`x` may have type `Y`
Expression `x === undefined` or `x !== undefined` inside a conditional	`x` may be `undefined`
Expression `x === null` or `x !== null` inside a conditional	`x` may be `null`
Expression `x == undefined`, `x != undefined`, `x == null`, or `x != null` inside a conditional	`x` may be `undefined` or `null`
Expression `x` or `!x` inside a conditional	`x` may be `undefined` or `null`
Assignment `x = undefined`	`x` may be `undefined`
Assignment `x = null`	`x` may be `null`
Assignment `z = x || y`	`x` may be `undefined` or `null`
Expression `x || {}` or `x || []`	`x` may be `undefined` or `null`

$(Q_2, \mathcal{T}_{d2})$, the following it true: Q_1 and Q_2 have a common, non-empty suffix, and $\mathcal{T}_{d1} = \mathcal{T}_{d2}$. This property is true for the two warnings of the above example, and TypeDevil merges them.

3) Merging by Array Type: TypeDevil reports multiple warnings if a single base type has multiple properties that have inconsistent types. This behavior can lead to many similar warnings for arrays that have multiple entries with inconsistent types. To avoid overwhelming developers with such warnings, TypeDevil merges two warnings $(t_1, p_1, \mathcal{T}_{i1})$ and $(t_2, p_2, \mathcal{T}_{i2})$ if t_1 and t_2 are the same array type.

4) Pruning via Belief Analysis: JavaScript code with intended polymorphism often uses runtime type checks to distinguish the types that a variable may have. For example, consider the code snippet in Figure 4. The first two arguments of `BigInteger()` are polymorphic and the function checks them for particular types. When a naive implementation of TypeDevil analyzes an execution where different types of arguments are given to `BigInteger()`, the analysis reports that the arguments of `BigInteger()` have inconsistent types, even though the program is correct.

To avoid reporting such false positives, we statically analyze the program to extract the programmer's "beliefs" [14] about types that a variable may have. The static analysis identifies common idioms used to check the runtime type of a variable and creates *type beliefs* for them (Table I). Each type belief consists of a variable name and a type that this variable is expected to have. TypeDevil leverages the inferred type beliefs to prune warnings where inconsistent types are expected by the programmer. To this end, TypeDevil associates each inferred type belief with the frame of the function where the variable in the belief is defined. To mark warnings for pruning, the analysis iterates through all warnings $(t, p, \mathcal{T}_i)$. If the base type t of a warning is a frame type, then TypeDevil removes all types t_i from $\mathcal{T}_i$ for which there exists a type belief saying that variable p may have type t_i. After this filtering of types, the analysis only keeps warnings with multiple remaining inconsistent types $\mathcal{T}_i$.

For Figure 4, TypeDevil infers the beliefs given in the comments and removes the expected types from any warnings for variables `a` and `b`. As a result, TypeDevil does not report a warning if `BigInteger()` is used correctly, but it would still report a warning if types that are not expected by the programmer are passed into the function.

5) Pruning by Degree of Inconsistency: To avoid false positives for highly polymorphic code, such as a generic function that operates on arbitrary data types, TypeDevil prunes a warning $(t, p, \mathcal{T}_i)$ if the number $|\mathcal{T}_i|$ of observed inconsistent types exceeds a threshold $maxTypes$. We use $maxTypes = 2$, and Section V-C evaluates the influence of this threshold.

6) Pruning by Size of Type Diff: As an additional technique to prune false positives caused by highly polymorphic code, TypeDevil prunes a warning if the size $|\mathcal{D}|$ of the type diff is larger than a threshold $maxTypeDiffs$. We use $maxTypeDiffs = 2$ and evaluate the threshold in Section V-C. The intuition behind this pruning technique is that variables and properties that contain very different types are likely to be polymorphic on purpose.

7) Pruning of Structural Subtypes: Subtype polymorphism is a kind of polymorphism that appears intentionally in many JavaScript programs. For example, libraries often mimic class-like inheritance via prototypes, and as a result, a reference may refer to either a supertype or a subtype. To avoid false positives for this kind of polymorphism, TypeDevil prunes a warning $(t, p, \mathcal{T}_i)$ if the types in $\mathcal{T}_i$ are in a structural subtyping relationship. This pruning step is in line with the definition of consistent types (Definition 2).

8) Pruning of `null`-related Warnings: Some programmers use `null` to indicate that a value is not available. Since `null` occurs in JavaScript only if the programmer explicitly uses it, TypeDevil does not consider types as inconsistent if these types are inconsistent because something can be `null` or have some other type. To remove `null`-related warnings, the analysis prunes a warning unless the warning's type diff contains a pair $(Q, \mathcal{T}_d)$ where `null` $\notin \mathcal{T}_d$.

V. EVALUATION

To evaluate the effectiveness of TypeDevil, we apply the approach to popular benchmarks and real-world web applications. In summary, the evaluation shows the following:

- TypeDevil effectively finds inconsistent types, many of which correspond to problems that programmers should

be aware of. In total, the analysis reports 33 warnings, of which at least 15 correspond to problematic code. (Section V-B)
- TypeDevil is complementary to JavaScript's strict mode, which warns about potential programming errors. None of the problems detected by our analysis are found with strict mode. (Section V-B)
- Our algorithm for condensing type graphs (Algorithm 1) effectively merges type nodes. On average, the algorithm reduces the number of type nodes to 66% of the initial number of type nodes, with a reduction of down to less than 1% for particular programs. (Table II)
- The techniques for merging and pruning warnings significantly reduce the number of warnings that TypeDevil reports, resulting in a number that is manageable in practice. The combination of all techniques presented in Section IV-C reduces 578 warnings to 33 warnings. (Section V-C)

A. Experimental Setup

Our implementation is open-source and available for download.[2] Both the dynamic analysis and the static belief analysis are implemented as source-to-source transformations. The dynamic analysis builds upon Jalangi [36]. To analyze web applications, we modify Spidermonkey (snapshot 151486), the JavaScript engine of Firefox, so that it intercepts all JavaScript code of web pages and passes it to our source-to-source transformations, before interpreting or compiling it.

Table II lists the programs we use to evaluate our approach, along with the number of non-comment, non-blank lines of JavaScript code (excluding third-party libraries). To measure lines of code we pre-process code with *js-beautify*,[3] which undoes the effects of compressing JavaScript code. We use all programs from the Sunspider benchmark suite and all but four programs from the Octane benchmark suite. We exclude Octane's *earley-boyer*, *typescript*, and *zlib* benchmarks because they consist of generated code. In contrast to human-written code, our observation that most JavaScript code follows implicit type rules does not hold for generated code, and as a result, TypeDevil reports an unusually high number of inconsistent types for these benchmarks. We exclude Octane's *code-load* because large parts of it are obfuscated, making it difficult for us to understand the type inconsistencies reported by TypeDevil. In addition to benchmark programs, we apply TypeDevil to seven real-world web applications. To exercise these applications, we load their initial page and perform a short sequence of typical user interactions. For all analyzed programs, we exclude code loaded from third-party libraries, such as JQuery and MooTools.

B. Type Inconsistencies Detected by the Analysis

In total, TypeDevil reports 33 inconsistent type warnings for the analyzed programs, out of which at least 15 correspond to problematic code. The last two columns of Table II lists the number of reported warnings for each program and how many of them we classify as problematic. the

TABLE II
SUMMARY OF RESULTS.

Program	LOC	Type graph nodes		Inconsistent types	
		Initially	Merged	All	True pos.
Sunspider:					
3d-cube	289	104	43	1	1
3d-morph	24	3	3	0	0
3d-raytrace	346	103	72	3	0
access-binary-trees	40	9	9	0	0
access-fannkuch	52	3	3	0	0
access-nbody	143	29	25	0	0
access-nsieve	29	6	6	0	0
bitops-3bit-bits-in-byte	17	5	5	0	0
bitops-bits-in-byte	18	5	5	0	0
bitops-bitwise-and	5	1	1	0	0
bitops-nsieve-bits	24	7	7	0	0
controlflow-recursive	18	7	7	0	0
crypto-aes	291	68	43	0	0
crypto-md5	213	32	32	1	1
crypto-sha1	148	26	26	1	1
date-format-tofte	211	53	47	4	0
date-format-xparb	379	38	37	1	1
math-cordic	57	13	12	0	0
math-partial-sums	25	3	3	0	0
math-spectral-norm	41	14	12	0	0
regexp-dna	1,697	14	6	1	1
string-base64	68	6	6	0	0
string-fasta	75	13	12	0	0
string-tagcloud	179	2,526	24	0	0
string-unpack-code	6	49	39	0	0
string-validate-input	75	13	13	0	0
Octane:					
box2d	10,880	1,438	1,333	0	0
crypto	1,873	295	282	1	0
deltablue	795	261	242	1	0
gbemu	9,771	1,443	1,416	6	5
mandreel	271,824	1,376	1,375	0	0
navier-strokes	676	150	142	1	0
pdfjs	55,182	171	164	0	0
raytrace	993	266	225	2	2
regexp	2,070	356	118	1	1
richards	616	168	160	0	0
splay	557	128	126	0	0
Web applications:					
annex	857	173	117	1	0
calculator	3,929	361	300	4	0
joomla	358	75	70	0	0
moodle	3,142	2,084	1,059	0	0
tenframe	1,276	187	166	0	0
todolist	5,938	964	914	4	2
zurmo	193	25	23	0	0
Total	375,400	13,071	8,730	33	15

1) Problematic Type Inconsistencies: The detected problems include the examples given in Table 1. The following describes additional representative problems detected by the analysis.

Out of bounds array access in Octane regexp: Octane's regexp benchmark has an out of bounds error that results in checking whether the `"undefined"` string matches a regular

[2]https://github.com/Berkeley-Correctness-Group/Jalangi-Berkeley
[3]https://github.com/beautify-web/js-beautify

expression. We reported this problem to the developers.[4] The root cause is that two arrays that are supposed to have the same length have a slightly different length. The problem does not crash the benchmark but leads to nonsensical computation. It is easy to fix by changing the length of one of the two arrays. TypeDevil detects this problem because an argument of a function contains `string` values most of the time but occasionally contains `undefined`.

Padding with `undefined` in Sunspider crypto-md5 and crypto-sha1: The two crypto benchmarks of Sunspider implement cryptographic algorithms that operate on blocks of a fixed length. To bring arbitrary input data to this length, the benchmarks add padding bytes. The ISO standard that specifies the padding mechanism [1] requires to add zero and one bits in particular ways. In addition to zeros and ones, the benchmark implementation creates blocks that also contain `undefined` values. As a result, several bitwise operations are performed on `undefined`, which happens to result in correct behavior due to type coercions. This type inconsistency is problematic for two reasons. First, the implementation deviates from the standard. Second, the implementation results in arrays that contain `undefined` values ("arrays with holes"), which is known to cause suboptimal performance because the JIT compiler cannot apply particular optimizations [17], [19]. TypeDevil detects these problems because functions that implement parts of the cryptographic algorithms have arguments that contain both `number` and `undefined` values.

Incorrect string literals in Todolist: The Todolist web application manages tasks and assigns each task to a part of the day. The possible parts are "morning", "afternoon", and "evening". At one code location, the programmer accidentally refers to "night" instead of "evening". Even though the code is clearly wrong, this inconsistency does not lead to obviously incorrect behavior because the code attempts to operate on non-existing DOM elements and therefore has no effect on the DOM. TypeDevil detects this problem because a function that is supposed to return the localized `string` value for "evening" returns `undefined` when being called with the incorrect value "night".

2) Additional Type Inconsistencies: Of the 33 warnings reported by TypeDevil, we classify 15 as certainly relevant for developers. The remaining warnings belong to two categories. First, several warnings point to code that is correct but that violates commonsense coding style guidelines. For example, these warnings include a method argument called "string" that sometimes contains a `number`, a variable that is initialized to a `number` even though it always contains values of type `array` in the remainder of the program, and polymorphic arrays, which are known to prevent particular optimizations. We do not classify these warnings as relevant because some developers may deem them as unimportant, but we nevertheless believe these warnings to be of value for other developers. Second, several warnings are false positives, which slip through the set of techniques for filtering warnings. For example, these warnings include polymorphic arguments of overloaded and generic functions.

3) Root Causes of Inconsistencies: The most prevalent root cause for inconsistencies is that the type of a variable, property, or function can be both `undefined` and some other type. For 16 of the 33 reported inconsistencies, the set of inconsistent types contains `undefined`. Despite this prevalence, simply reporting occurrences of `undefined` would significantly reduce the effectiveness of TypeDevil. In particular, such a simple analysis would miss four of the 15 warnings that correspond to problems because the set of inconsistent types of these warnings does not contain `undefined`. For example, these warnings include Figure 2b (an object type versus `string`) and Figure 2c (two inconsistent object types).

4) Comparison with Strict Mode: Strict mode [2] is a restricted variant of JavaScript that turns potential programming errors into exceptions. We compare TypeDevil to strict mode by running all programs from Table II in this mode. For 17 of the 44 programs, strict mode leads to an exception that is not raised in non-strict mode. Most of the exceptions are due to writing to an undeclared global variable, which is legal in JavaScript, but which may be unintended. None of the exceptions raised by strict mode points to a problem detected by TypeDevil. We conclude that our approach is complementary to strict mode.

C. Merging and Pruning of Warnings

The techniques for merging and pruning warnings effectively reduce the number of reported warnings from 578 to 33. To better understand the impact of each technique, Figure 5 shows the number of warnings that TypeDevil reports when particular techniques are enabled or disabled. For example, the figure shows that merging warnings by dataflow relations (Section IV-C1) reduces the number of warnings from 579 to 530, and that disabling this technique while enabling all others yields 42 warnings. The last box ("All techniques") shows the default configuration of TypeDevil, which we use throughout the paper. The results show that each technique contributes to reducing the number of reported warnings and that combining all techniques reduces the number the most. The last five columns analyze the impact of the two thresholds $maxTypes$ and $maxTypeDiffs$. The last column is the default configuration of TypeDevil, which we use throughout the rest of the paper.

VI. Related Work

Several approaches to infer and check types for programs written in (subsets) of JavaScript have been proposed [18], [20], [21], [37]. Heidegger and Thiemann's approach [20] infers flow-insensitive types similar to our definition of types. Guha et al.'s work [18] addresses the ubiquity of runtime type checks, which we address with a static belief analysis. DRuby [16] statically infers types for a subset of Ruby, and a dynamic analysis translates Ruby programs into this subset [15]. Rubydust [4] dynamically infers sound types when seeing all paths through a method, and it reports errors at

[4]https://code.google.com/p/octane-benchmark/issues/detail?id=21

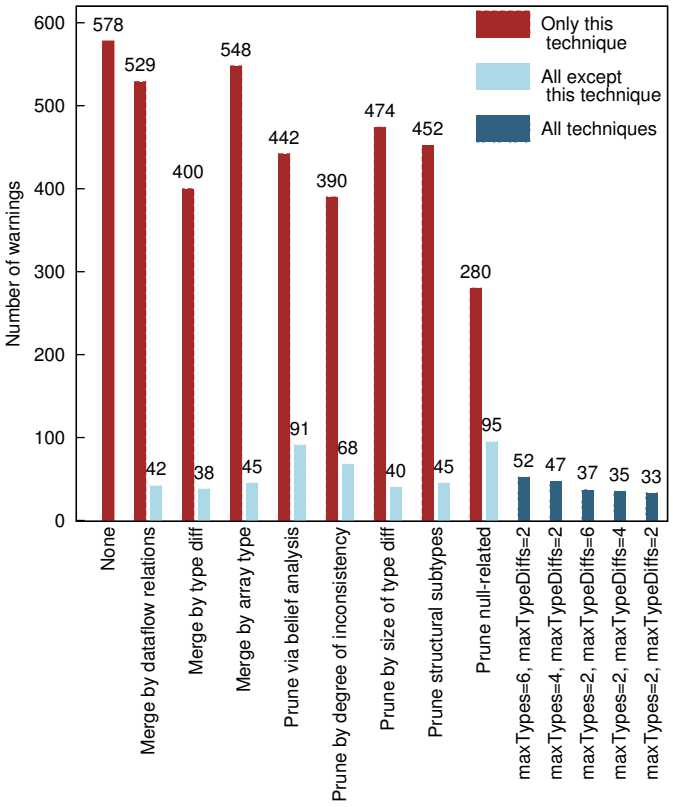


Fig. 5. Number of reported type inconsistencies depending on the techniques used to prune and merge warnings.

method boundaries. In contrast, TypeDevil also finds inconsistent types of local variables. All these approaches aim to show the well-typedness of a program, which leads to a high number of false positives due to the highly dynamic nature of real-world JavaScript code. For example, [15] perform 226 refactorings and add 177 type annotations to reduce false positives, which eventually leads to the discovery of 8 errors; [21] find that up to 39% of the checked code violates type properties (for the deltablue benchmark, which has one warning from TypeDevil).

Dynamic typing is also a performance challenge, and optimizing JIT compilers benefit from inferred types to emit type-specialized code [19], [22], [31]. Ahn et al. [3] improve the JavaScript engine's object representation to enable optimizations even though objects with different prototypes appear at a single code location. Our work avoids this problem by defining types independent of the prototype object. RPython is a statically typed subset of Python, which can be obtained from a subset of Python by partially executing the Python program [5].

TypeScript [9], [26] extends JavaScript with type annotations, which can prevent undesired type inconsistencies but also impose annotation effort onto developers. Doherty et al. [12] propose a static kind analysis for Matlab, which infers whether an identifier refers to a variable, a function, or the prefix of a package name. By searching for inconsistent types, TypeDevil also detects inconsistent kinds. Aycock [8] infers types for a subset of Python where each variable has a single type, which matches our observation that most code uses types consistently. A static analysis by Pradel et al. [29] detects type-related errors in Java programs by identifying method parameters that may hold values with unexpected types. Their work shares the idea to infer types from an existing program and to warn about inconsistencies.

Recent program analyses for JavaScript include information flow analyses [7], [10], dynamic determinacy analysis [35], symbolic execution approaches [34], and library-aware static analysis [23]. Several studies of real-world JavaScript code aim at better understanding the use of dynamic language features [33], in particular the notorious `eval()` [32], and the kinds of bugs found in JavaScript programs [27]. Based on their study of bugs, Ocariza et al. [28] propose a technique for finding fixes for particular kinds of bugs. To test JavaScript-based web applications, several approaches for generating GUI-level tests have been proposed [6], [11], [13], [24], [25], [30], [38]. These test generation approaches can be combined with TypeDevil to automatically explore and analyze complex applications.

Belief analysis has been proposed by Engler et al. [14] to find bugs that manifest as contradicting beliefs. Our work is the first to leverage belief analysis to remove likely false positives and to apply belief analysis in the context of a dynamically typed language.

VII. Conclusion

This paper presents TypeDevil, a mostly dynamic analysis to detect inconsistent types in JavaScript programs. The key idea is to gather type observations at runtime, to merge them into a graph, and to warn about variables, properties, and functions that have multiple inconsistent types. To deal with JavaScript code with intended polymorphism, the analysis aggressively prunes and merges warnings. TypeDevil takes a pragmatic compromise between proving the absence of type-related errors and finding problems with reasonable precision. We apply the approach to widely used benchmark programs and real-world web applications, leading to the discovery of 15 inconsistent types that correspond to problematic code.

The broader impact of this work is twofold. First, TypeDevil shows the power of dynamic analysis for finding problems in programs written in languages that are hard to analyze statically. We expect our results to encourage further research on dynamic analyses for JavaScript and other dynamic languages. Second, our work lays the foundation for a practical tool that helps JavaScript developers to detect type-related errors with minimal effort. We expect such a tool, if it is used during development, to find more problems than those we find in the well-tested programs used for the evaluation, and to significantly reduce the burden imposed on developers by the lack of static type checking.

ACKNOWLEDGMENT

This research is supported in part by NSF Grants CCF-0747390, CCF-1018729, CCF-1423645, and CCF-1018730, and gifts from Mozilla and Samsung.

REFERENCES

[1] Information technology – Security techniques – Message Authentication Codes (MACs) – Part 1: Mechanisms using a block cipher, ISO/IEC 9797-1:1999.

[2] ECMAScript language specification, 5.1 edition, June 2011.

[3] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas. Improving JavaScript performance by deconstructing the type system. In *PLDI*, 2014.

[4] J. D. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for Ruby. pages 459–472, 2011.

[5] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *DLS*, pages 53–64, 2007.

[6] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *ICSE*, pages 571–580, 2011.

[7] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS*, pages 113–124, 2009.

[8] J. Aycock. Aggressive type inference. In *International Python Conference*, 2000.

[9] G. M. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In *ECOOP*, pages 257–281, 2014.

[10] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *PLDI*, pages 50–62, 2009.

[11] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. WebMate: A tool for testing web 2.0 application. In *JSTools*, 2012.

[12] J. Doherty, L. J. Hendren, and S. Radpour. Kind analysis for MATLAB. In *OOPSLA*, pages 99–118, 2011.

[13] C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou. Ajax Crawl: Making Ajax applications searchable. In *ICDE*, pages 78–89, 2009.

[14] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

[15] M. Furr, J. D. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA*, pages 283–300, 2009.

[16] M. Furr, J. D. An, J. S. Foster, and M. W. Hicks. Static type inference for ruby. In *SAC*, pages 1859–1866, 2009.

[17] L. Gong, M. Pradel, and K. Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. Technical Report UCB/EECS-2014-144, EECS Department, University of California, Berkeley, Aug 2014.

[18] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *ESOP*, pages 256–275, 2011.

[19] B. Hackett and S. Guo. Fast and precise hybrid type inference for JavaScript. pages 239–250, 2012.

[20] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *ECOOP*, pages 200–224, 2010.

[21] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *SAS*, pages 238–255, 2009.

[22] F. Logozzo and H. Venter. RATA: Rapid atomic type analysis by abstract interpretation - application to JavaScript optimization. In *CC*, pages 66–83, 2010.

[23] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *ESEC/FSE*, pages 499–509, 2013.

[24] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *ICST*, pages 121–130, 2008.

[25] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling Ajax by inferring user interface state changes. In *ICWE*, pages 122–134, 2008.

[26] Microsoft. *TypeScript Language Specification, Version 1.0*. 2014.

[27] F. S. Ocariza Jr., K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side JavaScript bugs. pages 55–64, 2013.

[28] F. S. Ocariza Jr., K. Pattabiraman, and A. Mesbah. Vejovis: Suggesting fixes for JavaScript faults. In *ICSE*, pages 837–847, 2014.

[29] M. Pradel, S. Heiniger, and T. R. Gross. Static detection of brittle parameter typing. In *ISSTA*, pages 265–275, 2012.

[30] M. Pradel, P. Schuh, G. Necula, and K. Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. 2014.

[31] A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *POPL*, pages 481–494, 2012.

[32] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do - a large-scale study of the use of eval in JavaScript applications. In *ECOOP*, pages 52–78, 2011.

[33] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*, pages 1–12, 2010.

[34] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *S&P*, pages 513–528, 2010.

[35] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Dynamic determinacy analysis. In *PLDI*, pages 165–174, 2013.

[36] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE*, pages 488–498, 2013.

[37] P. Thiemann. Towards a type system for analyzing JavaScript programs. In *ESOP*, pages 408–422, 2005.

[38] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra. Guided test generation for web applications. In *ICSE*, pages 162–171, 2013.