DLint: Dynamically Checking Bad Coding Practices in JavaScript

Liang Gong[1], Michael Pradel[2], Manu Sridharan[3], and Koushik Sen[1]

[1] EECS Department, University of California, Berkeley, USA
[2] Department of Computer Science, TU Darmstadt, Germany, [3] Samsung Research America, USA
[1] {gongliang13, ksen}@cs.berkeley.edu
[2] michael@binaervarianz.de, [3] m.sridharan@samsung.com

ABSTRACT

JavaScript has become one of the most popular programming languages, yet it is known for its suboptimal design. To effectively use JavaScript despite its design flaws, developers try to follow informal code quality rules that help avoid correctness, maintainability, performance, and security problems. Lightweight static analyses, implemented in "lint-like" tools, are widely used to find violations of these rules, but are of limited use because of the language's dynamic nature. This paper presents DLint, a dynamic analysis approach to check code quality rules in JavaScript. DLint consists of a generic framework and an extensible set of checkers that each addresses a particular rule. We formally describe and implement 28 checkers that address problems missed by state-of-the-art static approaches. Applying the approach in a comprehensive empirical study on over 200 popular web sites shows that static and dynamic checking complement each other. On average per web site, DLint detects 49 problems that are missed statically, including visible bugs on the web sites of IKEA, Hilton, eBay, and CNBC.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.8 [**Software Engineering**]: Metrics

General Terms

Metrics, software development, software quality assurance

Keywords

Code practice, DLint, dynamic analysis, metric

1. INTRODUCTION

JavaScript has become one of the most popular programming languages. It powers various popular web sites, applications on mobile platforms, such as Firefox OS, Tizen OS, iOS, and Android, as well as desktop platforms, such as Windows 8 and Chrome OS. Despite its great success, JavaScript is not often considered a "well-formed" language. Designed and implemented in ten days,[1] JavaScript suffers from many unfortunate early design decisions that were preserved as the language thrived to ensure backward compatibility. The suboptimal design of JavaScript causes various pitfalls that developers should avoid [15].

A popular approach to help developers avoid common pitfalls are guidelines on which language features, programming idioms, APIs, etc. to avoid, or how to use them correctly. The developer community has learned such *code quality rules* over time, and documents them informally, e.g., in books [15, 28] and company-internal guidelines.[2] Following these rules improves software quality by reducing bugs, increasing performance, improving maintainability, and preventing security vulnerabilities. Since remembering and following code quality rules in JavaScript further burdens the use of an already complicated language, developers rely on automatic techniques that identify rule violations. The state-of-the-art approach for checking rules in JavaScript are lint-like static checkers [32], such as JSLint [3], JSHint [2], ESLint [1], and Closure Linter [4]. These static checkers are widely accepted by developers and commonly used in industry.

Although static analysis is effective in finding particular kinds of problems, it is limited by the need to approximate possible runtime behavior. Most practical static checkers for JavaScript [3, 2, 1, 4] and other languages [29, 11] take a pragmatic view and favor a relatively low false positive rate over soundness. As a result, these checkers may easily miss violations of some rules and do not even attempt to check rules that require runtime information.

Figure 1 shows two examples that illustrate the limitations of existing static checkers. The first example (Figure 1a) is a violation of the rule to not iterate over an array with a for-in loop (Section 2.4.1 explains the rationale for this rule). Existing static checkers miss this violation because they cannot precisely determine whether `props` is an array. The code snippet is part of *www.google.com/chrome*, which includes it from the Modernizr library. Because the code in Figure 1a misbehaves on Internet Explorer 7, the developers have fixed the problem in a newer version of the library.[3] The second example (Figure 1b) is a violation of the rule to avoid the notorious `eval` and other functions that dynamically evaluate code. The code creates an alias of `eval`, called `indirect`, and calls the `eval` function through this alias.

ISSTA'15, July 13–17, 2015, Baltimore, MD, USA
© 2015 ACM. 978-1-4503-3620-8/15/07...$15.00
http://dx.doi.org/10.1145/2771783.2771809

[1] http://brendaneich.com/2011/06/new-javascript-engine-module-owner/
[2] https://code.google.com/p/google-styleguide/
[3] https://github.com/Modernizr/Modernizr/pull/1419

```
// From Modernizr 2.6.2
for (i in props) {        // props is an array
  prop = props[i];
  before = mStyle.style[prop];
  ...
}
```

(a) Violation of the rule to avoid using for-in loops on an array. Found on *www.google.com/chrome*.

```
// From jQuery 2.1.0
globalEval: function(code) {
  var script, indirect = eval; // alias of eval function
  code = jQuery.trim( code );
  if (code) {
    if (code.indexOf("use strict") === 1) {
      ...
    } else {
      indirect(code);          // indirect call of eval
    }
  }
}
```

(b) Violation of the rule to avoid `eval` and its variants. Found on *www.repl.it*.

Figure 1: Examples that illustrate the limitations of static checking of code quality rules.

We found this example on *www.repl.it*, which includes the code from the jQuery library. Static checkers report direct calls of `eval` but miss indirect calls because static call resolution is challenging in JavaScript. Aliasing `eval` is used on various web sites [45] because it ensures that the code passed to `eval` is evaluated in the global context.

Despite the wide adoption of code quality rules and the limitations of static checking, there currently is no dynamic lint-like approach for JavaScript. This paper presents DLint, a dynamic analysis for finding violations of code quality rules in JavaScript programs. Our approach consists of a generic analysis framework and an extensible set of checkers built on top of the framework. We present 28 checkers that address common pitfalls related to inheritance, types, language usage, API usage, and uncommon values. We describe the checkers in a lightweight, declarative formalism, which yields, to the best of our knowledge, the first comprehensive description of dynamically checkable code quality rules for JavaScript. Some of the rules, e.g., Figure 1a, cannot be easily checked statically and are not addressed by existing static checkers. Other rules, e.g., Figure 1b, are addressed by existing static checkers, and DLint complements them through dynamic analysis.

Having DLint and the existing static checkers raises the following research questions, which compare the effectiveness of dynamic and static analyses:

- RQ1: How many violations of code quality rules are detected by DLint but missed by static checkers?
- RQ2: How many rule violations found by DLint are missed statically even though static checkers address the violated rule?

In addition, we also address this question:

- RQ3: How does the number of violations of code quality rules relate to the popularity of a web site?

To answer these questions, we perform an empirical study on over 200 of the world's most popular web sites. We apply DLint and the most widely adopted existing static checker, JSHint, to these sites and compare the problems they report with each other. In total, the study involves over 4 million lines of JavaScript code and 178 million covered runtime operations. The study shows that DLint identifies 53 rule violations per web site, on average, and that 49 of these warnings are missed by static checking (RQ1). Furthermore, we find that complementing existing static checkers with dynamic variants of these checkers reveals at least 10.1% additional problems that would be missed otherwise (RQ2). We conclude from these results that static and dynamic checking complement each other, and that pursuing the DLint approach is worthwhile.

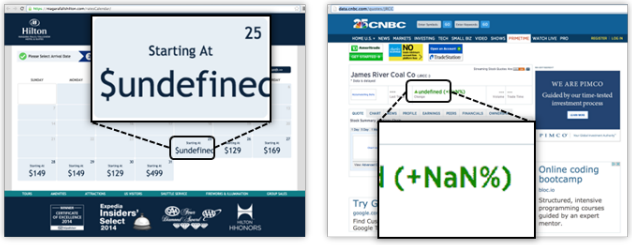


Figure 2: Bug found by DLint on the Hilton and CNBC web sites.

Even though this work is not primarily about bug finding (the rules we consider address a more diverse spectrum of code quality problems), we stumbled across 19 clear bugs in popular web sites when inspecting a subset of DLint's warnings. These bugs lead to incorrectly displayed web sites and are easily noticeable by users. For example, DLint detects "undefined" hotel rates on *www.hilton.com* and "NaN" values on *www.cnbc.com* (Figure 2). The approach also successfully identifies the motivating examples in Figure 1. All these examples are missed by static checking.

We envision DLint to be used as an in-house testing technique that complements static checkers in ensuring code quality. That is, our purpose is not to replace static lint-like tools but to provide an automatic approach for identifying problems missed by these tools. Our empirical results show that dynamic and static checking can each identify a unique set of violations of common code quality rules.

In summary, this paper contributes the following:

- We present the first dynamic analysis to find violations of code quality rules in JavaScript.
- We gather and formally describe 28 JavaScript quality rules that cannot be easily checked with static analysis.
- We present an extensive empirical study on over 200 popular web sites that systematically compares the effectiveness of static and dynamic analyses in finding code quality problems. The study shows that both approaches are complementary, and it quantifies their respective benefits.
- Our implementation of DLint can be easily extended with additional checkers, providing the basis for a practical tool that fills an unoccupied spot in the JavaScript tool landscape. DLint is available as open source: https://github.com/Berkeley-Correctness-Group/DLint

2. APPROACH

This section presents DLint, a dynamic analysis to detect violations of code quality rules. DLint consists of a

generic framework and a set of checkers that build upon the framework. Each checker addresses a rule and searches for violations of it. We specify when such a violation occurs in a declarative way through predicates over runtime events (Section 2.1). DLint currently contains 28 checkers that address rules related to inheritance (Section 2.2), types and type errors (Section 2.3), misuse of the JavaScript language (Section 2.4), misuse of an API (Section 2.5), and uncommon values (Section 2.6). The presented checkers reveal rule violations in various popular web sites (Section 4).

2.1 Rules, Events, and Runtime Patterns

The goal of this work is to detect violations of commonly accepted rules that the developer community has learned over time.

Definition 1 (Code quality rule)
A code quality rule is an informal description of a pattern of code or execution behavior that should be avoided or that should be used in a particular way. Following a code quality rule contributes to, e.g., increased correctness, maintainability, code readability, performance, or security.

We have studied 31 rules checked by JSLint [3], more than 150 rules checked by JSHint [2], and around 70 rules explained in popular guidelines [15, 28]. We find that existing static checkers may miss violations of a significant number of rules due to limitations of static analysis. Motivated by these findings, our work complements existing static checkers by providing a dynamic approach for checking code quality rules. To formally specify when a rule violation occurs, we describe violations in terms of predicates over events that happen during an execution.

Definition 2 (Runtime event predicate)
A runtime event predicate describes a set of runtime events with particular properties:

- $lit(val)$ *matches a literal, where the literal value is* val.
- $varRead(name, val)$ *matches a variable read, where the variable is called* $name$ *and has value* val.
- $call(base, f, args, ret, isConstr)$ *matches a function call, where the base object is* $base$*, the called function is* f*, the arguments passed to the function are* $args$*, the return value of the call is* ret*, and where* $isConstr$ *specifies whether the call is a constructor call.*
- $propRead(base, name, val)$ *matches a property read, where the base object is* $base$*, the property is called* $name$*, and the value of the property is* val.
- $propWrite(base, name, val)$ *matches a property write, where the base object is* $base$*, the property is called* $name$*, and the value written to the property is* val.
- $unOp(op, val, res)$ *matches a unary operation, where the operator is* op*, the input value is* val*, and the result value is* res.
- $binOp(op, left, right, res)$ *matches a binary operation, where the operator is* op*, the left and right operands are* $left$ *and* $right$*, respectively, and the result value is* res.
- $cond(val)$ *matches a conditional, where* val *is the value that is evaluated as a conditional.*
- $forIn(val)$ *matches a for-in loop that iterates over the object* val.

A predicate either constrains the value of a parameter or specifies with $*$ that the parameter can have any value. For example, $varRead(\text{“}foo\text{”}, *)$ is a runtime predicate that matches any read of a variable called “foo”, independent of the variable's value. The above list focuses on the events and parameters required for the runtime patterns presented in this paper. Our implementation supports additional events and parameters to enable extending DLint with additional checkers.

Based on runtime event predicates, DLint allows for specifying when a program violates a code quality rule during the execution. We call such a specification a *runtime pattern* and distinguish between two kinds of patterns:

Definition 3 (Single-event runtime pattern)
A single-event runtime pattern consists of one or more event predicates over a single runtime event, where each predicate is a sufficient condition for violating a code quality rule.

For example, $varRead(\text{“}foo\text{”}, *)$ is a single-event runtime patterns that addressed the trivial rule that no variable named “foo” should ever be read.

Definition 4 (Multi-event runtime pattern)
A multi-event runtime pattern consists of event predicates over two or more runtime events, if they occur together, are a sufficient condition for violating a code quality rule.

For example, $varRead(\text{“}foo\text{”}, *) \wedge varWrite(\text{“}foo\text{”}, *)$ is a multi-event runtime pattern that addresses the, again trivial, rule to not both read and write a variable named “foo” during an execution.

Because single-event runtime patterns match as soon as a particular event occurs, they can be implemented by a stateless dynamic analysis. In contrast, multi-event runtime patterns match only if two or more related events occur. Therefore, they require a stateful dynamic analysis.

The remainder of this section presents some of the code quality rules and their corresponding runtime patterns we address in DLint. Each rule is addressed by a checker that identifies occurrences of the runtime pattern. To the best of our knowledge, we provide the first comprehensive formal description of dynamic checks that address otherwise informally specified rules. Due to limited space, we discuss only a subset of all currently implemented checkers. The full list of checkers is available on our project homepage.

Abbreviations: $isFct(x)$, $isObject(x)$, $isPrim(x)$, and $isString(x)$ are true if x is a function, an object, a primitive value, and a string, respectively. $isArray(x)$, $isCSSObj(x)$, $isFloat(x)$, $isNumeric(x)$, $isBooleanObj(x)$, $isRegExp(x)$ are true if x is an array, a CSS object, a floating point value, a value that coerces into a number, a `Boolean` object, and a regular expression, respectively. $relOrEqOp$ refers to a relational operator or an equality operator. Finally, $argumentProps$ and $arrayProps$ refer to the set of properties of the built-in `arguments` variable and the set of properties in `Array.prototype`, respectively.

2.2 Problems Related to Inheritance

JavaScript's implementation of prototype-based inheritance not only offers great flexibility to developers but also provides various pitfalls that developers should avoid. To address some of these pitfalls, Table 2.1 shows DLint checkers that target inheritance-related rules. The following explains two of these checkers in detail.

Table 1: Inheritance-related code quality rules and runtime patterns (all are single-event patterns).

ID	Name	Code quality rule	Runtime event predicate(s)
I1	Enumerable-ObjProps	Avoid adding enumerable properties to `Object`. Doing so affects every for-in loop.	$propWrite(Object, *, *)$ $call(Object, f, args, *, *) \mid f.name = \text{“}defineProperty\text{”} \ \wedge$ $args.length = 3 \ \wedge \ args[2].enumerable = true$
I2	Inconsistent-Constructor	`x.constructor` should yield the function that has created `x`.	$propRead(base, constructor, val) \mid val \neq$ function that has created $base$
I3	NonObject-Prototype	The prototype of an object must be an object.	$propWrite(*, name, val) \mid name \in \{\text{“}prototype\text{”}, \text{“}\_proto\_\text{”}\} \ \wedge$ $!isObject(val)$
I4	Overwrite-Prototype	Avoid overwriting an existing prototype, as it may break the assumptions of other code.	$propWrite(base, name, *) \mid name \in \{\text{“}prototype\text{”}, \text{“}\_proto\_\text{”}\} \ \wedge$ $base.name$ is a user-defined prototype before the write
I5	Shadow-ProtoProp	Avoid shadowing a prototype property with an object property.	$propWrite(base, name, val) \mid val$ is defined in $base's$ prototype chain $\wedge$ $!isFct(val) \ \wedge \ (base, name) \notin shadowingAllowed$

Table 2: Code quality rules and runtime patterns related to type errors.

ID	Name	Code quality rule	Runtime event predicate(s)
Single-event patterns:			
T1	FunctionVs-Prim	Avoid comparing a function with a primitive.	$binOp(relOrEqOp, left, right, *) \mid isFct(left) \ \wedge \ isPrim(right)$ $binOp(relOrEqOp, left, right, *) \mid isPrim(left) \ \wedge \ isFct(right)$
T2	StringAnd-Undefined	Avoid concatenating a string and `undefined`, which leads to a string containing “undefined”.	$binOp(+, left, right, res) \mid (left = \text{“}undefined\text{”} \ \vee \ right = \text{“}undefined\text{”}) \ \wedge$ $isString(res)$
T3	ToString	`toString` must return a string.	$call(*, f, *, ret, *) \mid f.name = \text{“}toString\text{”} \ \wedge \ !isString(ret)$
T4	Undefined-Prop	Avoid accessing the “undefined” property.	$propWrite(*, \text{“}undefined\text{”}, *)$ $propRead(*, \text{“}undefined\text{”}, *)$
Multi-event patterns:			
T5	Constructor-Functions	Avoid using a function both as constructor and as non-constructor.	$call(*, f, *, *, false) \ \wedge \ call(*, f, *, *, true)$
T6	TooMany-Arguments	Pass at most as many arguments to a function as it expects.	$call(*, f, args, *, *) \mid \lvert args\rvert > f.length \ \wedge$ $\nexists \ varRead(arguments, *)$ during the call

2.2.1 Inconsistent Constructor

Each object has a `constructor` property that is supposed to return the constructor that has created the object. Unfortunately, JavaScript does not enforce that this property returns the constructor, and developers may accidentally set this property to arbitrary values. The problem is compounded by the fact that all objects inherit a `constructor` property from their prototype.

For example, consider the following code, which mimics class-like inheritance in an incorrect way:

```
1 function Super() {}  // superclass constructor
2 function Sub() {     // subclass constructor
3   Super.call(this);
4 }
5 Sub.prototype = Object.create(Super.prototype);
6 // Sub.prototype.constructor = Sub; // should do this
7 var s = new Sub();
8 console.log(s.constructor);          // "Function: Super"
```

Because the code does not assign the correct constructor function to `Sub`'s prototype, accessing the constructor of an instance of `Sub` returns `Super`.

To detect such inconsistent constructors, DLint checks for each read of the `constructor` property whether the property's value is the base object's constructor function (Checker I2 in Table 2.1). To access the function that has created an object, our implementation stores this function in a special property of every object created with the `new` keyword.

2.2.2 Shadowing Prototype Properties

Prototype objects can have properties, which are typically used to store data shared by all instances of a prototype. Developers of Java-like languages may think of prototype properties as static, i.e., class-level, fields. In such languages, it is forbidden to have an instance field with the same name as an existing static field. In contrast, JavaScript does not warn developers when an object property shadows a prototype property. However, shadowing is discouraged because developers may get easily confused about which property they are accessing.

To identify shadowed prototype properties, Checker I5 in Table 2.1 warns about property writes where the property is already defined in the base object's prototype chain. For example, the following code raises a warning:

```
1 function C() {}; C.prototype.x = 3;
2 var obj = new C(); obj.x = 5;
3 console.log(obj.x);  // "5"
```

There are two common and harmless kinds of violations of this rule in client-side JavaScript code: changing prototype properties of DOM objects (e.g., `innerHTML`), and overriding of functions inherited from the prototype object. To avoid overwhelming developers with unnecessary warnings, the checker excludes a set *shadowingAllowed* of such DOM properties and properties that refer to functions.

2.3 Problems Related to Types

JavaScript does not have compile time type checking and is loosely typed at runtime. As a result, various problems that would lead to type errors in other languages may remain unnoticed. Table 2.1 shows DLint checkers that warn about such problems by checking type-related rules. Two of these checkers require to check for occurrences of multi-event

Table 3: Code quality rules and runtime patterns related to language misuse (all are single-event patterns).

ID	Name	Code quality rule	Runtime event predicate(s)
L1	Arguments-Variable	Avoid accessing non-existing properties of `arguments`.	$propRead(arguments, name, *) \mid name \notin argumentProps$ $propWrite(arguments, *, *)$ $call(arguments, f, *, *, *) \mid f.name =$ "$concat$"
L2	ForInArray	Avoid for-in loops over arrays, both for efficiency and because it may include properties of `Array.prototype`.	$forIn(val) \mid isArray(val)$
L3	GlobalThis	Avoid referring to `this` when it equals to `global`.	$varRead(this, global)$
L4	Literals	Use literals instead of `new Object()` and `new Array()`[1]	$call(builtin, f, args, *, *) \mid (f = Array \vee f = Object) \wedge args.length = 0$
L5	NonNumeric-ArrayProp	Avoid storing non-numeric properties in an array.	$(propWrite(base, name, *) \vee propRead(base, name, *)) \mid isArray(base) \wedge !isNumeric(name) \wedge name \notin arrayProps)$
L6	PropOf-Primitive	Avoid setting properties of primitives, which has no effect.	$propWrite(base, *, *) \mid isPrim(base)$

[1] Note that it is legitimate for performance reasons to call these constructors with arguments [24].

runtime patterns. We explain two type-related checkers in the following.

2.3.1 Accessing the "undefined" Property

An object property name in JavaScript can be any valid JavaScript string. As developers frequently store property names in variables or in other properties, this permissiveness can lead to surprising behavior when a property name coerces to "undefined". For example, consider the following code:

```
var x;                    // undefined
var y = {}; y[x] = 23;    // results in { undefined: 23 }
```

The undefined variable `x` is implicitly converted to the string "undefined". Developers should avoid accessing the "undefined" property because it may result from using an undefined value in the square bracket notation for property access. Checker T4 checks for property reads and writes where the property name equals "undefined".

2.3.2 Concatenate `undefined` and a String

JavaScript allows programs to combine values of arbitrary types in binary operations, such as + and −. If differently typed operands are combined, the JavaScript engine implicitly converts one or both operands to another type according to intricate rules [5]. Even though such type coercions may often match the intent of the programmer [42], they can also lead to hard to detect, incorrect behavior.

A rare and almost always unintended type coercion happens when a program combines an uninitialized variable and a string with the + operator. In this case, JavaScript coerces `undefined` to the string "undefined" and concatenates the two strings.

2.4 Problems Related to Language Misuse

Some of JavaScript's language features are commonly misunderstood by developers, leading to subtle bugs, performance bottlenecks, and unnecessarily hard to read code. DLint checks several rules related to language misuse (Table 2.2.2), three of which we explain in the following.

2.4.1 For-in Loops over Arrays

JavaScript provides different kinds of loops, including the for-in loop, which iterates over the properties of an object. For-in loops are useful in some contexts, but developers are discouraged from using for-in loops to iterate over arrays. The rationale for this rule is manifold. For illustration, consider the following example, which is supposed to print "66":

```
var sum = 0, x, array = [11, 22, 33];
for (x in array) {
    sum += array[x];
}
console.log(sum);
```

First, because for-in considers all properties of an object, including properties inherited from an object's prototype, the iteration may accidentally include enumerable properties of `Array.prototype`. E.g., suppose a third-party library expands arrays by adding a method: `Array.prototype.m = ...;`. In this case, the example prints "`66function () {...}`". Some browsers, e.g., Internet Explorer 7, mistakenly iterate over all built-in methods of arrays, causing unexpected behavior even if `Array.prototype` is not explicitly expanded. Second, some developers may incorrectly assume that a for-in loop over an array iterates through the array's elements, similar to, e.g., the for-each construct in Java. In this case, a developer would replace the loop body from above with `sum += x`, which leads to the unexpected output "`0012`". Finally, for-in loops over arrays should be avoided because they are significantly slower than traditional for loops.[4]

Checker L2 helps avoiding these problems by warning about for-in loops that iterate over arrays. Given DLint's infrastructure, this checker boils down to a simple check of whether the value provided to a for-in loop is an array.

2.4.2 Properties of Primitives

When a program tries to access properties or call a method of one of the primitive types `boolean`, `number`, or `string`, JavaScript implicitly converts the primitive value into its corresponding wrapper object. For example:

```
var fact = 42;
fact.isTheAnswer = true;
```

Unfortunately, setting a property of a primitive does not have the expected effect because the property is attached to a wrapper object that is immediately discarded afterwards. In the example, the second statement does not modify `fact` but a temporarily created instance of `Number`, and `fact.isTheAnswer` yields `undefined` afterwards.

[4] E.g., V8 refuses to optimize methods that include for-in loops over arrays.

Table 4: Code quality rules and runtime patterns related to incorrect API usage (single-event patterns).

ID	Name	Code quality rule	Runtime event predicate(s)
A1	Double-Evaluation	Avoid `eval` and other ways of runtime code injection.	$call(builtin, eval, *, *, *)$ $call(builtin, Function, *, *, *)$ $call(builtin, setTimeout, args, *, *) \mid isString(args[0])$ $call(builtin, setInterval, args, *, *) \mid isString(args[0])$ $call(document, f, *, *, *) \mid f.name =$ "$write$"
A2	EmptyChar-Class	Avoid using an empty character class, `[]`, in regular expressions, as it does not match anything.	$lit(val) \mid isRegExp(val) \wedge val$ contains "[]" $call(builtin, RegExp, args, *, *) \mid isString(args[0]) \wedge args[0]$ contains "[]"
A3	FunctionTo-String	Avoid calling `Function.toString()`, whose behavior is platform-dependent.	$call(base, f, *, *, *) \mid f.name =$ "$toString$" $\wedge\ isFct(base)$
A4	FutileWrite	Writing a property should change the property's value.	$propWrite(base, name, val) \mid base[name] \neq val$ after the write
A5	MissingRadix	Pass a radix parameter to `parseInt`, whose behavior is platform-dependent otherwise.	$call(builtin, parseInt, args, *, *) \mid args.length = 1$
A6	SpacesIn-Regexp	Prefer " `{N}`"2 over multiple consecutive empty spaces in regular expressions for readability.	$lit(val) \mid isRegExp(val) \wedge val$ contains " " $call(builtin, RegExp, args, *, *) \mid args[0]$ contains " "
A7	StyleMisuse	CSS objects are not strings and should not be used as if they were.	$binOp(eqOp, left, right) \mid isCSSObj(left) \wedge isString(right)$ $binOp(eqOp, left, right) \mid isString(left) \wedge isCSSObj(right)$
A8	Wrapped-Primitives	Beware that all wrapped primitives coerce to `true`.	$cond(val) \mid isBooleanObj(val) \wedge val.valueOf() = false$

Developers can prevent such surprises by following the rule that setting properties of primitives should be avoided. Checker L6 checks for violations of this rule by warning about every property write event where the base value is a primitive.

2.4.3 *Unnecessary Reference to* `this`

The semantics of `this` in JavaScript differ from other languages and often confuse developers. When accessing `this` in the context of a function, the value depends on how the function is called. In the global context, i.e., outside of any function, `this` refers to the global object. Because the global object is accessible without any prefix in the global context, there is no need to refer to `this`, and a program that accesses `this` in the global context is likely to confuse the semantics of `this`. Checker L3 warns about accesses of `this` in the global context by checking whether reading `this` yields the global object.

2.5 Problems Related to API Misuse

As most APIs, JavaScript's built-in API and the DOM API provide various opportunities for misusing the provided functionality. Motivated by commonly observed mistakes, several DLint checkers address rules related to incorrect, unsafe, or otherwise discouraged API usages (Table 2.4). The following explains three checkers in detail.

2.5.1 *Coercion of Wrapped Primitives*

The built-in constructor functions `Boolean`, `Number`, and `String` enable developers to wrap primitive values into objects. However, because objects always coerce to `true` in conditionals, such wrapping may lead to surprising behavior when the wrapped value coerces to `false`. For example, consider the following example, where the code prints "true" in the second branch, even though `b` is `false`:

```
1 var b = false;
2 if (b) console.log("true");
3 if (new Boolean(b)) console.log("true");
```

To avoid such surprises, developers should avoid evaluating wrapped boolean in conditionals. Checker A8 warns about code where a `Boolean` object appears in a conditional and where the value wrapped by the object is `false`.

2.5.2 *Futile Writes of Properties*

Some built-in JavaScript objects allow developers to write a particular property, but the write operation has no effect at runtime. For example, typed arrays[5] simply ignore all out-of-bounds writes. Even though such futile writes are syntactically correct, they are likely to be unintended and, even worse, difficult to detect because JavaScript silently executes them.

Checker A4 addresses futile writes by warning about property write operations where, after the write, the base object's property is different from the value assigned to the property. In particular, this check reveals writes where the property remains unchanged. An alternative way to check for futile writes is to explicitly search for writes to properties that are known to not have any effect. The advantage of the runtime predicate we use is to provide a generic checker that detects all futile writes without requiring an explicit list of properties.

2.5.3 *Treating* `style` *as a String*

Each DOM element has a `style` attribute that determines its visual appearance. The `style` attribute is a string in HTML, but the `style` property of an HTML DOM element is an object in JavaScript. For example, the JavaScript DOM object that correspond to the HTML markup `<div style='top:10px;'></div>` is a CSS object with a property named `top`. The mismatch between JavaScript and HTML types sometimes causes confusion, e.g., leading to JavaScript code that retrieves the `style` property and compares it to a string. Checker A7 identifies misuses of the `style` property by warning about comparison operations, e.g., `===` or `!==`, where one operand is a CSS object and where the other operand is a string.

[5]Typed arrays are array-like objects that provide a mechanism for accessing raw binary data stored in contiguous memory space.

Table 5: Code quality rules and runtime patterns related to uncommon values (all are single-event patterns).

ID	Name	Code quality rule	Runtime event predicate(s)
V1	Float-Equality	Avoid checking the equality of similar floating point numbers, as it may lead to surprises due to rounding.[2]	$binOp(eqOp, left, right, *) \mid isFloat(left) \wedge isFloat(right) \wedge |left - right| < \epsilon$
V2	NaN	Avoid producing `NaN` (not a number).	$unOp(*, val, NaN) \mid val \neq NaN$ $binOp(*, left, right, NaN) \mid left \neq NaN \wedge right \neq NaN$ $call(*, *, args, NaN, *) \mid NaN \notin args$
V3	Overflow-Underflow	Avoid numeric overflow and underflow.	$unOp(*, val, \infty) \mid val \neq \infty$ $binOp(*, left, right, \infty) \mid left \neq \infty \wedge right \neq \infty$ $call(builtin, *, args, \infty, *) \mid \infty \notin args$

[2] It is a notorious fact that the expression `0.1 + 0.2 === 0.3` returns `false` in JavaScript.

2.6 Problems Related to Uncommon Values

The final group of checkers addresses rules related to uncommon values that often occur unintendedly (Table 2.5.1). We explain one of them in detail.

2.6.1 Not a Number

The `NaN` (not a number) value may result from exceptional arithmetic operations and is often a sign of unexpected behavior. In JavaScript, `NaN` results not only from operations that produce `NaN` in other languages, such as division by zero, but also as the result of unusual type coercions. For example, applying an arithmetic operation to a non-number, such as `23-"five"`, may yield `NaN`. Since generating `NaN` does not raise an exception or any other kind of warning in JavaScript, `NaN`-related problems can be subtle to identify and hard to diagnose.

In most programs, developers want to follow the rule that suggests to avoid occurrences of `NaN`. Checker V2 warns about violations of this rule by identifying operations that take non-`NaN` values as inputs and that produce a `NaN` value. The checker considers unary and binary operations as well as function calls.

3. IMPLEMENTATION

We implement DLint as an automated analysis tool for JavaScript-based web applications and node.js applications. The system has multiple steps. First, DLint opens a web site in Firefox, which we modify so that it intercepts all JavaScript code before executing it, including code dynamically evaluated through `eval`, `Function`, `setInterval`, `setTimeout`, and `document.write`. Second, DLint instruments the intercepted code to add instructions that perform the checks. This part of DLint builds upon the dynamic analysis framework Jalangi [49]. Third, while the browser loads and renders the web site, the instrumented code is executed and DLint observes its runtime events. If an event or a sequence of events matches a runtime pattern, DLint records this violation along with additional diagnosis information. Fourth, after completely loading the web site, DLint automatically triggers events associated with visible DOM elements, e.g., by hovering the mouse over an element. This part of our implementation builds upon Selenium.[6] We envision this step to be complemented by a UI-level regression test suite, by manual testing, or by a more sophisticated automatic UI testing approach [6, 13, 47]. Finally, DLint gathers the warnings from all checkers and reports them to the developer. Our prototype implementation has around 12,000 lines of JavaScript, Java and Bash code, excluding projects we build upon. The implementation is available as open-source.

A key advantage of DLint is that the framework can easily be extended with additional dynamic checkers. Each checker registers for particular runtime events and gets called by the framework whenever these events occur. The framework dispatches events to an arbitrary number of checkers and hides the complexity of instrumentation and dispatching. For example, consider the execution of `a.f=b.g`. DLint instruments this statement so that the framework dispatches the following four events, in addition to executing the original code: $varRead(\text{“}b\text{”}, x_1)$, $propRead(x_1, \text{“}g\text{”}, x_2)$, $varRead(\text{“}a\text{”}, x3)$, $propWrite(x3, \text{“}f\text{”}, x2)$, where x_i refer to runtime values.

4. EVALUATION

We evaluate DLint through an empirical study on over 200 web sites. Our main question is whether dynamically checking for violations of code quality rules is worthwhile. Section 4.2 addresses this question by comparing DLint to a widely used static code quality checker. Section 4.3 explores the relationship between code quality and the popularity of a web site. We evaluate the performance of DLint in Section 4.4. Section 4.5 presents examples of problems that DLint reveals. Finally, Section 4.6 discusses threats to the validity of our conclusions.

4.1 Experimental Setup

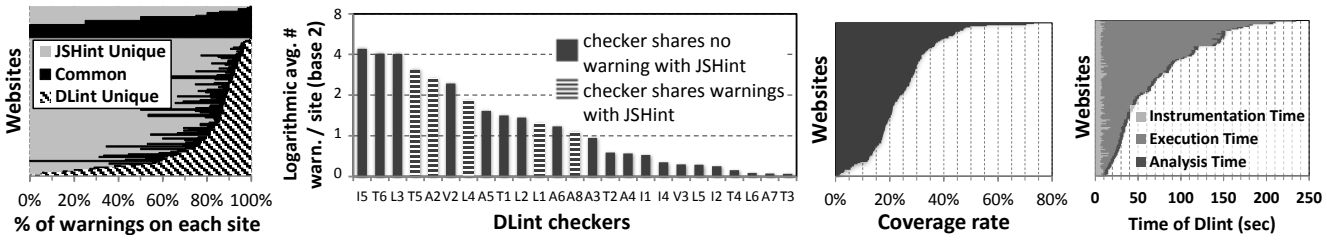
The URLs we analyze come from two sources. First, we analyze the 50 most popular web sites, as ranked by Alexa. Second, to include popular web sites that are not landing pages, we search Google for trending topics mined from Facebook and include the URLs of the top ranked results. In total, the analyzed web sites contain 4 million lines of JavaScript code. Since many sites ship minified source code, where an entire script may be printed on a single line, we pass code to js-beautify[7] before measuring the number of lines of code. We fully automatically analyze each URL as described in Section 3.

To compare DLint to static checking, we analyze all code shipped by a web site with JSHint. To the best of our knowledge, JSHint is currently the most comprehensive and widely used static, lint-like checker for JavaScript.[8] We compare the problems reported by DLint and JSHint through an AST-based analysis that compares the reported code locations and the kinds of warnings.

[6] http://www.seleniumhq.org/

[7] http://jsbeautifier.org/

[8] JSHint checks more code quality rules than JSLint. ESLint is a re-implementation of JSLint to support pluggable checkers.



(a) Warn. # per site. (b) Avg. warn. # from DLint per site. (c) Coverage rate. (d) Running time.

Figure 3: Warnings from JSHint and DLint.

4.2 Dynamic versus Static Checking

DLint checks 28 rules, of which 5 have corresponding JSHint checkers. JSHint checks 150 rules, of which 9 have corresponding DLint checkers. There is no one-to-one mapping of the overlapping checkers. For example, DLint's "DoubleEvaluation" checker (Checker A1 in Table 2.4) corresponds to several JSHint checkers that search for calls of `eval` and `eval`-like functions. In total over all 200 web sites analyzed, DLint reports 9,018 warnings from 27 checkers, and JSHint reports about 580k warnings from 91 checkers. That is, JSHint warns about significantly more code quality problems than DLint. Most of them are syntactical problems, such as missing semicolons, and therefore are out of the scope of a dynamic analysis. For a fair comparison, we focus on JSHint checkers that have a corresponding DLint checker.

To further compare the state-of-the-art static checker and DLint, we design research Questions RQ1 and RQ2 and answer those questions through empirical studies. RQ1 studies the number of additional violations detected by dynamic analysis in general. RQ2 studies the number of violations that are meant to be detected by static checkers but are actually missed by JSHint in practice.

RQ1: How many violations of code quality rules are detected by DLint *but missed by static checkers?*

Figure 3a shows for each analyzed web site the percentage of warnings reported only by JSHint, by both DLint and JSHint, and only by DLint. Each horizontal line represents the distribution of warnings for a particular web site. The results show that DLint identifies warnings missed by JSHint for most web sites and that both checkers identify a common set of problems.

To better understand which DLint checkers contribute warnings that are missed by JSHint, Figure 3b shows the number of warnings reported by all DLint checkers, on average per web site. The black bars are for checkers that report problems that are completely missed by JSHint. These checkers address rules that cannot be easily checked through static analysis. The total number of DLint warnings per site ranges from 1 to 306. On average per site, DLint generates 53 warnings, of which 49 are problems that JSHint misses.

In both RQ1 and RQ2, warnings from JSHint and DLint are matched based on their reported code locations. For the same code practice violation, there are sometimes slight differences (different column offset) between the locations reported by the two systems. To improve the matching precision, we first approximately match warnings reported on the same lines; then predefined rules are applied to prune impossible warning matchings (e.g., `eval` warnings from JSHint can only match warnings from checker Checker A1 in DLint); finally, we manually inspect all matches to check their validity.

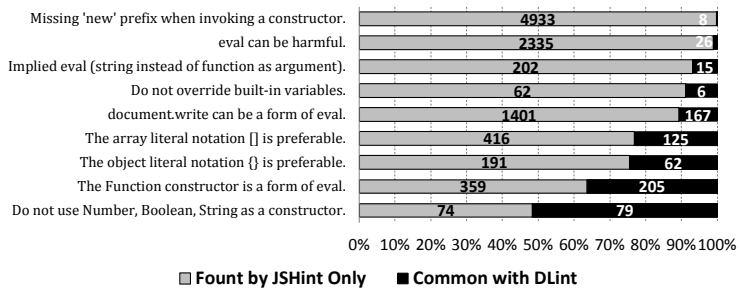
RQ2: How many rule violations found by DLint *are missed statically even though static checkers address the violated rule?*

One of the motivations of this work is that a pragmatic static analysis may miss problems even though it searches for them. In RQ2, we focus on DLint checkers that address rules that are also checked by JSHint and measure how many problems are missed by JSHint but revealed by DLint. Figure 4a (4b) shows the number of warnings detected by JSHint (DLint) checkers that address a rule also checked by DLint (JSHint). The figure shows that JSHint and DLint are complementary. For example, JSHint and DLint both detect 205 calls of `Function`, which is one form of calling the evil `eval`. JSHint additionally detects 359 calls that are missed by DLint because the call site is not reached at runtime. DLint additionally detects 181 calls of `eval` and `eval`-like functions, which includes calls of `Function`.

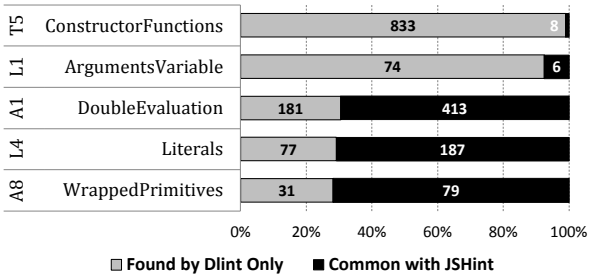
Considering all checkers shown in Figure 4, DLint reports 10.1% additional warnings that are missed by JSHint even though JSHint checks for the rule. Manual inspection of these problems shows that they are due to code that is hard or impossible to analyze for a pragmatic static checker, e.g., code that assigns `Function` to another variable and calls it through this alias.

We conclude from the results for RQ1 and RQ2 that dynamically checking for violations of code quality rules is worthwhile. DLint complements existing static checkers by revealing problems that are missed statically and by finding violations of rules that cannot be easily checked through static analysis.

Coverage: Dynamic analysis is inherently limited to the parts of a program that are covered during an execution. To understand the impact of this limitation, we compute coverage as the number of basic operations that are executed at least once divided by the total number of basic operations. For example, `if(b) a=1` consists of three basic operations: read `b`, test whether `b` evaluates to `true`, and write `a`. If during the execution, `b` always evaluated to `false`, then the coverage rate would be $2/3 = 66.7\%$. Figure 3c shows the coverage achieved during the executions that DLint analyzes. Each line represents the coverage rate of one web site. Overall, coverage is relatively low for most web sites, suggesting that, given a richer test suite, DLint could reveal additional rule violations.



(a) Warnings from JSHint that have a matching warning from DLint.



(b) Warnings from DLint that have a matching warning from JSHint.

Figure 4: Overlap of warnings reported by DLint and JSHint.

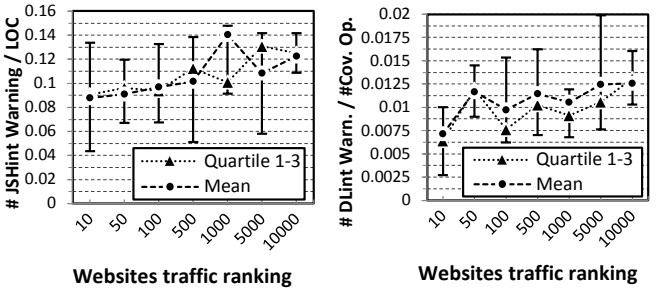


Figure 5: Number of warnings over site rank.

Figure 6: `NaN` and overflow bugs found by DLint.

4.3 Code Quality versus Web Site Popularity

RQ3: How does the number of violations of code quality rules relate to the popularity of a web site?

We address this question by analyzing web sites of different popularity ranks and by computing the overall number of warnings produced by DLint and JSHint. More specifically, we gather a random sample of web sites from the Alexa top 10, 50, ..., 10000 web sites and apply both checkers to each set of samples. Figure 5 shows the number of reported warnings. The left figure shows the relation between the number of JSHint warnings reported per LOC (vertical axis) and the ranking of the web site (horizontal axis). The right figure shows the relation between the number of DLint warnings per operation executed (vertical axis) and the ranking of the web site (horizontal axis). The correlation is 0.6 and 0.45 for DLint and JSHint, respectively, suggesting that popular web sites tend to have fewer violations of code quality rules.

4.4 Performance of the Analysis

Figure 3d shows the overall time required to apply DLint to a web site. The time is composed of the time to intercept and instrument JavaScript code, the time to execute code and render the page, and the time to summarize and report warnings. Most of the time is spent executing the instrumented code, which naturally is slower than executing non-instrumented code [49]. Given that DLint is a fully automatic testing approach, we consider the performance of our implementation to be acceptable.

4.5 Examples of Detected Problems

The main goal of checking code quality rules is not to detect bugs but to help developers avoid potential problems. Nevertheless, we were happy to stumble across 19 serious problems, such as corrupted user interfaces and displaying incorrect data, while inspecting warnings reported by DLint. Due to limited space, this section reports only some examples, all of which are missed by JSHint.

Not a Number. Checker V2 reveals several occurrences of `NaN` that are visible on the web site. For example, Figure 6 shows `NaN` bugs detected by DLint on the web sites of IKEA (1) and eBay (2), where articles cost the incredibly cheap "$NaN", and an official website of a basketball team[9] (3), where a player had "NaN" steals and turnovers. To help understand the root cause, DLint reports the code location where a `NaN` originates. For example, the IKEA web site loads the data to display and dynamically insert the results into the DOM. Unfortunately, some data items are missing and the JavaScript code initializes the corresponding variables with `undefined`, which are involved in an arithmetic operation that finally yields `NaN`.

Overflows and Underflows. Figure 6 (4 and 5) shows two bugs related to arithmetic overflow and underflow detected by DLint on the sites of Tackle Warehouse and CCNEX. The cause of the problem are arithmetic operations that yield an infinite value that propagates to the DOM.

Futile Write. DLint warns about the following code snippet on Twitch, a popular video streaming web site:

```
1 window.onbeforeunload=
2   "Twitch.player.getPlayer().pauseVideo();"
3 window.onunload="Twitch.player.getPlayer().pauseVideo();"
```

The code attempts to pause the video stream when the page is about to be closed. Unfortunately, these two event handlers are still *null* after executing the code, because de-

[9] http://www.uconnhuskies.com/

velopers must assign a function object as an event handler of a DOM element. Writing a non-function into an event handler property is simply ignored by DOM implementations.

Style Misuse. DLint found the following code on Craigslist:

```
1 if (document.body.style === "width:100\%") { ... }
```

The developer tries to compare `style` with a string, but `style` is an object that coerces into a string that is meaningless to compare with, e.g., "CSS2Properties" in Firefox.

Properties of Primitives. Besides web sites, we also apply DLint to the Sunspider and Octane benchmark suites. Due to space limitations, detailed results are omitted. The following is a problem that Checker L6 detects in Octane's GameBoy Emulator benchmark:

```
1 var decode64 = "";
2 if (dataLength > 3 && dataLength % 4 == 0) {
3   while (index < dataLength) {
4     decode64 += String.fromCharCode(...)
5   }
6   if (sixbits[3] >= 0x40) {
7     decode64.length -= 1; // writing a string's property
8   }
9 }
```

Line 7 tries to remove the last character of `decode64`. Unfortunately, this statement has no side effect because the `string` primitive is coerced to a `String` object before accessing `length`, leaving the `string` primitive unchanged.

4.6 Threats to Validity

The validity of the conclusions drawn from our results are subject to several threats. First, both DLint and JSHint include a limited set of checkers, which may or may not be representative for dynamic and static analyses that check code quality rules in JavaScript. Second, since DLint and JSHint use different reporting formats, our approach for matching the warnings from both approaches may miss warnings that refer to the same problem and may incorrectly consider reports as equivalent. We carefully inspect and revise the matching algorithm to avoid such mistakes.

5. RELATED WORK

Several dynamic analyses for JavaScript have been proposed recently, e.g., to detect type inconsistencies [41], data races [44], and cross-browser issues [38]. JITProf identifies performance bottlenecks caused by code that JIT engines cannot effectively optimize [24]. Each of these approaches addresses a particular kind of error, whereas DLint is a generic framework for checking code quality rules. Other dynamic program analyses for JavaScript include determinacy analysis [48], information flow analyses [14, 7], library-aware static analysis [33], and symbolic execution approaches [47]. BEAR [23] allows for inferring user interaction requirements from runtime traces. In contrast, our work aims at finding violations of coding practices.

Static analyses beyond the checkers discussed elsewhere in this paper include analyses to detect potential type errors [53, 31, 27, 25] and a sophisticated points-to analysis [52]. Feldthaus et al. propose a refactoring framework for specifying and enforcing JavaScript practices [20]. JSNose [19] is a metric-based technique that can detect 13 code smells in JavaScript. In contrast to these approaches, our work explores how dynamic analysis can complement existing static analyses. Several authors propose approaches for semi-automatically [36, 22] or automatically [12] repairing JavaScript applications.

Artzi et al. propose a UI-level test generation framework for web applications [6]. EventBreak [40] uses a performance-guided test generation algorithm to identify unresponsiveness web applications. These and other UI-level test generation approaches [37, 34, 18, 6, 17, 54, 13] may be combined with DLint to extend the set of analyzed executions.

The complementary nature of static and dynamic analysis can be exploited by combining both approaches [51]. Feldthaus et al. combine dynamic and static analysis of JavaScript to check if TypeScript [39] interfaces match a library's implementation [21]. DSD-Crasher [16] analyses static code and runtime behaviour to find bugs. Others combine both approaches to find security vulnerabilities [26] and to debug and repair faults in Java programs [50].

FindBugs [30] is a static checker of code quality rules in Java, which is similar in spirit to the checkers we compare with and which also has been widely used in industry [10, 9]. PQL [35] is a "program query language" designed to express code quality rules that programmers should follow. It focuses on rules that can be expressed with a subset of the runtime events supported by DLint. For example, PQL cannot express queries over unary and binary operations.

Several empirical studies to understand the abilities of bug detection tools have been performed. Rutar et al. compare static bug finding tools for Java [46]. Rahman et al. compare static bug checkers and defect prediction approaches [43]. Ayewah et al.[8] study and discuss the warnings reported by FindBugs [30]. Our work differs from these studies by directly comparing static and dynamic analyses for JavaScript.

6. CONCLUSION

This paper describes DLint, a dynamic analysis that consists of an extensible framework and 28 checkers that address problems related to inheritance, types, language misuse, API misuse, and uncommon values. Our work contributes the first formal description of these otherwise informally documented rules and the first dynamic checker for rule violations. We apply DLint in a comprehensive empirical study on over 200 of the world's most popular web sites and show that dynamic checking complements state-of-the-art static checkers. Static checking misses at least 10.1% of the problems it is intended to find. Furthermore, DLint addresses problems that are hard or impossible to reveal statically, leading to 49 problems, on average per web site, that are missed statically but found by DLint. Since our approach scales well to real-world web sites and is easily extensible, it provides a first step in filling an currently unoccupied spot in the JavaScript tool landscape.

7. ACKNOWLEDGEMENTS

This research is supported by NSF Grants CCF-1423645 and CCF-1409872, by a gift from Mozilla, by a Sloan Foundation Fellowship, by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, and by the German Research Foundation (DFG) within the Emmy Noether Project "ConcSys".

8. REFERENCES

[1] ESLint. http://eslint.org/.

[2] JSHint. http://jshint.com/.

[3] JSLint. http://www.jslint.com/.

[4] The Closure Linter enforces the guidelines set by Google. https://code.google.com/p/closure-linter/.

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

[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] N. Ayewah, W. P. J., D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Workshop on Program analysis for software tools and engineering*, 2007.

[9] N. Ayewah and W. Pugh. The google findbugs fixit. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 241–252, 2010.

[10] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Using findbugs on production software. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 805–806, 2007.

[11] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[12] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 237–246, 2010.

[13] W. Choi, G. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[14] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 50–62. ACM, 2009.

[15] D. Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.

[16] C. Csallner, Y. Smaragdakis, and T. Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

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

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

[19] A. M. Fard and A. Mesbah. JSNOSE: detecting javascript code smells. In *13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2013, Eindhoven, Netherlands, September 22-23, 2013*, pages 116–125, 2013.

[20] A. Feldthaus, T. D. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for javascript. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 119–138, 2011.

[21] A. Feldthaus and A. Møller. Checking correctness of typescript interfaces for javascript libraries. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 1–16. ACM, 2014.

[22] J. Frolin S. Ocariza, K. Pattabiraman, and A. Mesbah. Vejovis: Suggesting fixes for JavaScript faults. In *ICSE*, 2014.

[23] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining behavior models from user-intensive web applications. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 277–287, 2014.

[24] 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.

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

[26] W. G. J. Halfond, S. R. Choudhary, and A. Orso. Improving penetration testing through static and dynamic analysis. *Softw. Test., Verif. Reliab.*, 21(3):195–214, 2011.

[27] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 200–224, 2010.

[28] D. Herman. *Effective JavaScript: 68 Specific ways to harness the power of JavaScript*. Addison-Wesley, 2013.

[29] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 132–136. ACM, 2004.

[30] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[31] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Symposium on Static Analysis (SAS)*, pages 238–255. Springer, 2009.

[32] S. C. Johnson. Lint, a C program checker, 1978.

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

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

[35] M. C. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: A program query language. In *Conference on*

Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 365–383. ACM, 2005.

[36] F. Meawad, G. Richards, F. Morandat, and J. Vitek. Eval begone!: semi-automated removal of eval from javascript programs. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 607–620, 2012.

[37] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling Ajax by inferring user interface state changes. In *International Conference on Web Engineering (ICWE)*, pages 122–134, 2008.

[38] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 561–570, 2011.

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

[40] M. Pradel, P. Schuh, G. Necula, and K. Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014.

[41] M. Pradel, P. Schuh, and K. Sen. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *International Conference on Software Engineering (ICSE)*, 2015.

[42] M. Pradel and K. Sen. The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[43] F. Rahman, S. Khatri, E. T. Barr, and P. T. Devanbu. Comparing static bug finders and statistical prediction. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 424–434, 2014.

[44] V. Raychev, M. T. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 151–166, 2013.

[45] 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.

[46] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 245–256. IEEE Computer Society, 2004.

[47] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *IEEE Symposium on Security and Privacy*, pages 513–528, 2010.

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

[49] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013.

[50] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold. Fault localization and repair for java runtime exceptions. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, pages 153–164, 2009.

[51] Y. Smaragdakis and C. Csallner. Combining static and dynamic reasoning for bug detection. In *International Conference on Tests and Proofs (TAP)*, pages 1–16. Springer, 2007.

[52] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of javascript. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, pages 435–458, 2012.

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

[54] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra. Guided test generation for web applications. In *International Conference on Software Engineering (ICSE)*, pages 162–171. IEEE, 2013.