DLint: Dynamically Checking Bad Coding Practices in JavaScript

Liang Gong¹, Michael Pradel², Manu Sridharan³ and Koushik Sen¹

¹ UC Berkeley ² TU Darmstadt ³ Samsung Research America
Why JavaScript?

- The RedMonk Programming Language Rankings (1st) — Based on GitHub and StackOverflow
- Web assembly language
- Web applications, DSL, Desktop App, Mobile App
Problematic JavaScript

- Designed and Implemented in 10 days →
- Not all decisions were well thought →
- **Problematic language features**
  - Error prone
  - Inefficient code
  - Security loophole
- Problematic features are retained
  - backward compatibility
Problematic JavaScript
What is coding practice?

• Good coding practices
  • informal rules
  • improve quality

• Better quality means:
  • Less correctness issues
  • Better performance
  • Better usability
  • Better maintainability
  • Less security loopholes
  • Less surprises
  • ...
Rule: avoid using `for..in` over arrays

```javascript
var sum = 0, value;
var array = [11, 22, 33];
for (value in array) {
    sum += value;
}
> sum ?
```
Rule: avoid using `for..in` over arrays

```javascript
var sum = 0, value;
var array = [11, 22, 33];
for (value in array) {
    sum += value;
}
> sum ?
```

- 11 + 22 + 33 => 66 (not array value)
- 0 + 1 + 2 => 3
- 0+"0"+"1"+"2" => "0012"
Rule: avoid using `for...in` over arrays

```javascript
var sum = 0, value;
var array = [11, 22, 33];
for (value in array) {
    sum += value;
}
> sum ?
```

- 11 + 22 + 33 => 66 (not array value)
- 0 + 1 + 2 => 3
- 0+"0"+"1"+"2" => "0012"

- Cross-browser issues
- Result depends on the Array prototype object
Rule: avoid using `for..in` over arrays

```javascript
var sum = 0, value;
var array = [11, 22, 33];
for (value in array) {
  sum += value;
}
> sum ?

for (i=0; i < array.length; i++) {
  sum += array[i];
}

function addup(element, index, array) {
  sum += element;
}
array.forEach(addup);
```
Rule: avoid using `for..in` over arrays

```javascript
var sum = 0, value;
var array = [11, 22, 33];
for (value in array)
    sum += value;
> sum ?

for (i=0; i < array.length; i++)
    sum += array[i];

function addup(element, index, array)
    sum += element;
array.forEach(addup);
```
Coding Practices and Lint Tools

• **Existing Lint-like checkers**
  – Inspect source code
  – Rule-based checking
  – Detect common mistakes
  – Enforce coding conventions

• **Limitations:**
  – Approximates behavior
  – Unknown aliases
  – Lint tools favor precision over soundness

• **Difficulty:** Precise static program analysis
DLint

- Dynamic Linter checking code quality rules for JS
- Open-source, robust and extensible framework
- Formalized and implemented 28 rules
  - Counterparts of static rules
  - Additional rules
- Empirical study
  - Compare static and dynamic checking
Jalangi: A Dynamic Analysis Framework for JavaScript
Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs

\[
\begin{align*}
    a.f &= b.g & \rightarrow & \text{PutField}(&\text{Read}("a", a), "f", \text{GetField}(\text{Read}("b", b), "g")) \\
    \text{if} (a.f()) &\ldots & \rightarrow & \text{if} (&\text{Branch}(\text{Method}(\text{Read}("a", a), "f"))()) &\ldots \\
    x &= y + 1 & \rightarrow & x &= \text{Write}("x", \text{Binary}(‘+’, \text{Read}("y", y), \text{Literal}(1)))
\end{align*}
\]

\[
\begin{align*}
    \text{analysis.Literal}(c) & & \text{analysis.Branch}(c) \\
    \text{analysis.Read}(n, x) & & \text{analysis.Write}(n, x) \\
    \text{analysis.PutField}(b, f, v) & & \text{analysis.Binary}(op, x, y) \\
    \text{analysis.Function}(f, \text{isConstructor}) & & \text{analysis.GetField}(b, f) \\
    \text{analysis.Method}(b, f, \text{isConstructor}) & & \text{analysis.Unary}(op, x) \\
    \ldots & & \ldots
\end{align*}
\]
Runtime Patterns

- Single-event: **Stateless** checking
- Multi-event: **Stateful** checking
Language Misuse

Avoid setting properties of primitives, which has no effect.

```javascript
var fact = 42;
fact.isTheAnswer = true;
console.log(fact.isTheAnswer);
> undefined
```

DLint Checker Predicate:

\[ propWrite(base, *, *) \land isPrim(base) \]
Uncommon Values

Avoid producing NaN (Not a Number).

```javascript
var x = 23 - "five";
> NaN
```

DLint Checker Predicate:

- `unOp(*, val, NaN) ∧ val ≠ NaN`
- `binOp(*, left, right, NaN) ∧ left ≠ NaN ∧ right ≠ NaN`
- `call(*, *, args, NaN, *) ∧ NaN ≠ args`
Avoid concatenating undefined to string.

```javascript
var value;
...
var str = "price: ";
...
var result = str + value;
> "price: undefined"

DLint Checker Predicate:

\[ binOp(+, left, right, res) \]
\[ \land (left = undefined \lor right = undefined) \]
\[ \land \text{isString}(res) \]
API Misuse

Beware that all wrapped primitives coerce to true.

```javascript
var b = false;
if (new Boolean(b)) {
    console.log("true");
}
> true
```

DLint Checker Predicate:

\[ cond(val) \land \text{isWrappedBoolean}(val) \land \text{val.valueOf}() = false \]
### Table 1: Inheritance-related code quality rules and runtime patterns (all are single-event patterns).

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Code quality rule</th>
<th>Runtime event predicate(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>Enumerable-ObjProps</td>
<td>Avoid adding enumerable properties to Object. Doing so affects every for-in loop.</td>
<td><code>propWrite(Object, *, *)</code> &lt;br&gt;`call(Object, f, args, *, *)</td>
</tr>
<tr>
<td>I2</td>
<td>Inconsistent-Constructor</td>
<td><code>x.constructor</code> should yield the function that has created x.</td>
<td>`propWrite(base, constructor, val)</td>
</tr>
<tr>
<td>I3</td>
<td>NonObject-Prototype</td>
<td>The prototype of an object must be an object.</td>
<td>`propWrite(*, name, val)</td>
</tr>
<tr>
<td>I4</td>
<td>Overwrite-Prototype</td>
<td>Avoid overwriting an existing prototype, as it may break the assumptions of other code.</td>
<td>`propWrite(base, name, *, *)</td>
</tr>
<tr>
<td>I5</td>
<td>Shadow-ProtoProp</td>
<td>Avoid shadowing a prototype property with an object property.</td>
<td>`propWrite(base, name, val)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Code quality rule</th>
<th>Runtime event predicate(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single-event patterns:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>FunctionVs-Prim</td>
<td>Avoid comparing a function with a primitive.</td>
<td>`binOp(relOrEqOp, left, right, *)</td>
</tr>
<tr>
<td>T2</td>
<td>StringAnd-Undef</td>
<td>Avoid concatenating a string and undefined, which leads to a string containing “undefined”.</td>
<td>`binOp(+, left, right, res)</td>
</tr>
<tr>
<td>T3</td>
<td>ToString</td>
<td><code>toString</code> must return a string.</td>
<td>`call(*, f, *, ret, *)</td>
</tr>
<tr>
<td>T4</td>
<td>Undefined-Prop</td>
<td>Avoid accessing the “undefined” property.</td>
<td><code>propWrite(*, “undefined”, *)</code> &lt;br&gt;<code>propRead(*, “undefined”, *)</code></td>
</tr>
<tr>
<td><strong>Multi-event patterns:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>Constructor-Functions</td>
<td>Avoid using a function both as constructor and as non-constructor.</td>
<td><code>call(*, f, *, *, false) ∧ call(*, f, *, *, true)</code></td>
</tr>
<tr>
<td>T6</td>
<td>TooMany-Arguments</td>
<td>Pass at most as many arguments to a function as it expects.</td>
<td>`call(*, f, args, *, *)</td>
</tr>
<tr>
<td>ID</td>
<td>Name</td>
<td>Code quality rule</td>
<td>Runtime event predicate(s)</td>
</tr>
<tr>
<td>----</td>
<td>--------------------------</td>
<td>-----------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>A1</td>
<td>Double-Evaluation</td>
<td>Avoid <code>eval</code> and other ways of runtime code injection.</td>
<td><code>call(builtin, eval, *, *, *, *)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>call(builtin, Function, *, *, *, *)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>`call(builtin, setTimeout, args, *, *, *)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>`call(builtin, setInterval, args, *, *, *)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>`call(document, f, *, *, *, *)</td>
</tr>
<tr>
<td>A2</td>
<td>EmptyChar-Class</td>
<td>Avoid using an empty character class, <code>[]</code>, in regular expressions, as it does not match anything.</td>
<td>`lit(val)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>`call(builtin, RegExp, args, *, *)</td>
</tr>
<tr>
<td>A3</td>
<td>FunctionToString</td>
<td>Avoid calling <code>Function.toString()</code>, whose behavior is platform-dependent.</td>
<td>`call(base, f, *, *, *, *)</td>
</tr>
<tr>
<td>A4</td>
<td>FutileWrite</td>
<td>Writing a property should change the property’s value.</td>
<td>`propWrite(base, name, val)</td>
</tr>
<tr>
<td>A5</td>
<td>MissingRadix</td>
<td>Pass a radix parameter to <code>parseInt</code>, whose behavior is platform-dependent otherwise.</td>
<td>`call(builtin, parseInt, args, *, *)</td>
</tr>
<tr>
<td>A6</td>
<td>SpacesIn-Regexp</td>
<td>Prefer “{N}”2 over multiple consecutive empty spaces in regular expressions for readability.</td>
<td>`lit(val)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>`call(builtin, RegExp, args, *, *)</td>
</tr>
<tr>
<td>A7</td>
<td>StyleMisuse</td>
<td>CSS objects are not strings and should not be used as if they were.</td>
<td>`binOp(eqOp, left, right)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>`binOp(eqOp, left, right)</td>
</tr>
<tr>
<td>A8</td>
<td>Wrapped-Primitives</td>
<td>Beware that all wrapped primitives coerce to <code>true</code>.</td>
<td>`cond(val)</td>
</tr>
</tbody>
</table>
Table 3: Code quality rules and runtime patterns related to language misuse (all are single-event patterns).

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Code quality rule</th>
<th>Runtime event predicate(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>Arguments-Variable</td>
<td>Avoid accessing non-existing properties of arguments.</td>
<td>propRead(arguments, name, *)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>propWrite(arguments, *, *)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>call(arguments, f, *, *, *)</td>
</tr>
<tr>
<td>L2</td>
<td>ForInArray</td>
<td>Avoid for-in loops over arrays, both for efficiency and because it may include properties of Array.prototype.</td>
<td>forIn(val)</td>
</tr>
<tr>
<td>L3</td>
<td>GlobalThis</td>
<td>Avoid referring to this when it equals to global.</td>
<td>varRead(this, global)</td>
</tr>
<tr>
<td>L4</td>
<td>Literals</td>
<td>Use literals instead of new Object() and new Array() (^1)</td>
<td>call(builtin, f, args, *, *)</td>
</tr>
<tr>
<td>L5</td>
<td>NonNumeric-ArrayProp</td>
<td>Avoid storing non-numeric properties in an array.</td>
<td>(propWrite(base, name, *) ( \lor ) propRead(base, name, *))</td>
</tr>
<tr>
<td>L6</td>
<td>PropOf-Primitive</td>
<td>Avoid setting properties of primitives, which has no effect.</td>
<td>propWrite(base, *, *)</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Code quality rule</th>
<th>Runtime event predicate(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>Float-Equality</td>
<td>Avoid checking the equality of similar floating point numbers, as it may lead to surprises due to rounding. (^2)</td>
<td>binOp(eqOp, left, right, *)</td>
</tr>
<tr>
<td>V2</td>
<td>NaN</td>
<td>Avoid producing NaN (not a number).</td>
<td>unOp(*, val, NaN)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>binOp(*, left, right, NaN)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>call(*, *, args, NaN, *)</td>
</tr>
<tr>
<td>V3</td>
<td>Overflow-Underflow</td>
<td>Avoid numeric overflow and underflow.</td>
<td>unOp(*, val, ( \infty ))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>binOp(*, left, right, ( \infty ))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>call(builtin, *, args, ( \infty ), *, *)</td>
</tr>
</tbody>
</table>

\(^2\) It is a notorious fact that the expression \(0.1 + 0.2 = 0.3\) returns false in JavaScript.
**DLint Overview**

- **Instrument SpiderMonkey** to intercept JavaScript files
- **Transpile JavaScript code with Jalangi** [Sen et al. FSE 2013]
- **DLint checks runtime states** and find issues
- **Report** reason and code location
Evaluation

Research Questions
• DLint warning vs. JSHint warning?
• Additional warnings from DLint?
• Coding convention vs. page popularity?

Experimental Setup
• 200 web sites (top 50 + others)
• Comparison to JSHint
Some sites: One approach finds all
Most sites: Better together

% of Warnings: DLint vs. JSHint
Additional Warnings Reported by DLint

- 53 warnings per page
- 49 are missed by JSHint
Correlation between Alexa popularity and number of DLint warnings: 0.6
ALGOT
Frame/6 wire baskets/top shelf
$64
Last year's price: $NaN

ALGOT
Frame/wire baskets/rod
$204
Last year's price: $220

ALGOT
Frame/4 wire baskets/top shelf
$44
Last year's price: $48

ALGOT
Frame/4 wire baskets/top shelf
$60
Last year's price: $NaN

ALGOT
Frame/4 wire baskets
$35
Last year's price: $39

ALGOT
Frame/4 wire baskets
$51
Last year's price: $NaN

ALGOT
Frame/4 mesh baskets/top shelf
$56
Last year's price: $60

ALGOT
Frame with rod/wire baskets
$74
Last year's price: $87
Rule: avoid **setting field on primitive values**

From Google Octane Game Boy Emulator benchmark:

```javascript
var decode64 = "";
if (dataLength > 3 && dataLength % 4 == 0) {
    while (index < dataLength) {
        decode64 += String.fromCharCode(...);
    }
    if (sixbits[3] >= 0x40) {
        decode64.length -= 1;
    }
}
```
Rule: avoid **setting field on primitive values**

From Google Octane Game Boy Emulator benchmark:

```javascript
var decode64 = "";
if (dataLength > 3 && dataLength % 4 == 0) {
    while (index < dataLength) {
        decode64 += String.fromCharCode(...);
    }
    if (sixbits[3] >= 0x40) {
        decode64.length -= 1;
    }
}
```

No effect because `decode64` is a primitive string.
Rule: avoid no effect operations

```javascript
window.onbeforeunload=
"Twitch.player.getPlayer().pauseVideo();"

window.onunload=
"Twitch.player.getPlayer().pauseVideo();"
```
Rule: avoid no effect operations

⚠️ `window.onbeforeunload=`
"Twitch.player.getPlayer().pauseVideo();"

⚠️ `window.onunload=`
"Twitch.player.getPlayer().pauseVideo();"

```javascript
window.onbeforeunload = function () {
    Twitch.player.getPlayer().pauseVideo();
}
```
Takeaways

Dynamic lint-like checking for JavaScript

- **Static checkers are not sufficient**, DLint complements
- **DLint** is a open-source, robust and extensible tool
  - Works on **real-world websites**
  - Found **19 clear bugs on most popular websites**

More information:

- **Paper**: “DLint: Dynamically Checking Bad Coding Practices in JavaScript”
- **Source Code**: https://github.com/Berkeley-Correctness-Group/DLint
- **Google** “DLint Berkeley”
Takeaways

Dynamic lint-like checking for JavaScript

- Static checkers are not sufficient, DLint complements
- DLint is a open-source, robust and extensible tool
  - Works on real-world websites
  - Found 19 clear bugs on most popular websites

More information:
- Source Code: https://github.com/Berkeley-Correctness-Group/DLint
- Google “DLint Berkeley”

Thanks!
Formalization: **declarative specification**

1. Predicates over runtime events

   - `propWrite(base, name, val)`
   - `propRead(base, name, val)`
   - `cond(val)`
   - `unOp(op, val, res)`
   - `binOp(op, left, right, res)`
   - `call(base, f, args, ret, isConstr)`

**Example:**

```
propWrite(*, "myObject", val)
| isPrim(val)
```
Missing 'new' prefix when invoking a constructor.

eval can be harmful.

Implied eval (string instead of function as argument).

Do not override built-in variables.
document.write can be a form of eval.

The array literal notation [] is preferable.
The object literal notation {} is preferable.
The Function constructor is a form of eval.

Do not use Number, Boolean, String as a constructor.
E.g., 181 calls of eval(), Function(), etc. missed by JSHint
Description

Infinity% - -Infinity% Off
MSRP: $Infinity - $Infinity
You Save: $Infinity - $Infinity

Built using only top-of-the-line components, the Fat Sack Tackle Company Fizzle Jig delivers a high level of attraction and weedless performance. It goes where other vibrating jigs can't, to catch bass that other jigs won't.

Fitted with a matching hex blade, the Fat Sack Tackle Company Fizzle Jig produces a strong vibration and an eye-catching flash. The premium hex blade also pulls double duty as a weedguard to keep the hook point from snagging.

Customer Reviews
hot sellers

- StrapsCo Genuine Patent Leather Watch Strap Women's Band in Black
- 18mm Shark Mesh Stainless Steel Watch Band Strap fits Breitling
- 20mm Shark Mesh Stainless Steel Watch Band Strap fits Breitling
- 22mm Shark Mesh Stainless Steel Watch Band Strap fits Breitling
- Shark Mesh Watch Band Strap Breitling Navitimer Superocean 18mm
- Matte Black PVD Shark Mesh Watch Band Strap fits Seiko 18mm 20mm
- Yellow Gold PVD Shark Mesh Watch Band Strap fits Breitling 18mm 20mm
- StrapsCo Expansion Watch Band Stainless Steel Strap sz 12mm
<prices>
  <normal>
    <priceNormal unformatted="9.9">$9.90</priceNormal>
    <pricePrevious />
    <priceNormalPerUnit />
    <pricePreviousPerUnit />
  </normal>
</prices>

previousPrice = getElementValue("pricePrevious" + suffix, normal);

parseFloat(previousPrice).toFixed(2).replace(…)
  .replace('.00', '');

XML: Empty Element

JS: undefined

JS: NaN
Type Related Checker

Avoid accessing the undefined property.

```javascript
var x; // undefined
var y = {};
⚠️
y[x] = 23; // { undefined: 23 }

propWrite(*, "undefined", *)
propRead(*, "undefined", *)
```
Chained Analysis

\[ a.f = b.g \]

\[ \text{PutField}(\text{Read}("a", a), "f", \text{GetField}(\text{Read}("b", b), "g")) \]
Rule: avoid using `for..in` on arrays

www.google.com/chrome,
included code from Modernizr:

```
for (i in props) {  // props is an array
    prop = props[i];
    before = mStyle.style[prop];
    ...
}
```
**API Misuse**

`eval` is evil, do not use `eval`.

```javascript
var fun = eval;
...
⚠️ `fun("var a = 1;")`;
```

- `call(builtin, eval, *, *, *)`
- `call(builtin, Function, *, *, *)`
- `call(builtin, setTimeout, args, *, *)`  
  | `isString(args[0])`
- `call(builtin, setInterval, args, *, *)`  
  | `isString(args[0])`
- `call(document, write, *, *, *)`
YOUR CONFERENCE PRESENTATION

HOW YOU PLANNED IT:

START
INTRODUCE YOURSELF
DESCRIBE OUTLINE OF TALK

MOTIVATION
RESULTS

15 MINUTES
METHODOLOGY AND EXPERIMENT DESIGN
CONCLUSIONS

HOW IT GOES:

PREVIOUS SPEAKER RUNS LATE AND EATS INTO YOUR TIME.

START

TECHNICAL DIFFICULTIES CONNECTING YOUR LAPTOP.

FORGET INTRODUCING YOURSELF.

SPEND WAY TOO MUCH TIME DESCRIBING YOUR OUTLINE.

ANNYING AUDIENCE MEMBER INTERRUPTS WITH SELF-AGGRAVATING QUESTION.

REALIZE YOU ONLY HAVE 3 MINUTES LEFT.

POWER THROUGH THE REST OF YOUR 30 SLIDES.

15 MINUTES

Awkward silence Q&A.