Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries

Authored by M Madsen, B Livshits, M Fanning
Aarhus University & Microsoft Research
ESEC / FSE 2013

Presented by Jing Pu
Outline

- Motivation
- Challenges
- Approach & Key Techniques
- Evaluation
- Conclusion
Motivation

- **Research target**
  JavaScript applications execute in a rich execution environment
  - web programs
  - Server-side programs

- **Problem**
  Library and OS invocation codes are ignored and not well analyzed.

- **How to in-depth statically analyze:**
  JavaScript applications in the windows 8 OS?
Win 8 JavaScript applications

- This is the composition of a typical Windows 8 JavaScript application.
- Large size of library objects.
- Depends on libraries communicating with HTML DOM
- Uses Windows Runtime libraries
- Used built-in DOM API and other popular libraries and frameworks.
Challenges

- Rely on environment libraries
  - Browser API
  - HTML DOM
  - Invoke OS libraries at Windows runtime

- Popular libraries reflective JavaScript features
  - Reflective calls
  - Eval
  - Computed properties
  - Runtime modification of properties

- Reason about the objects information return from libraries & pass into callbacks
Approach & Key Techniques

- **Approach**
  
  Infer what the *objects* are based on *observing uses of library functionality* within application code.

- **Key Techniques**
  - Pointer analysis
  - Use analysis
Examples

- **Example 1**: DOM-manipulating code snippet

```javascript
var canvas = document.querySelector("#leftcol .logo");
var context = canvas.getContext("2d");
context.fillRect(20, 20, c.width / 2, c.height / 2);
context.strokeRect(0, 0, c.width, c.height);
```

**Q**: What object does `querySelector` return?

**A**: `HTMLCanvasElement`

**M**: Use pointer analysis & use analysis
Examples

Example 2: Stubs from WinRT library

```javascript
    writeAsync = function() {{},
    flushAsync = function() {{},
    close = function() {} }
}
```

```javascript
var s = Windows.Storage.Stream;
var fs = new s.FileOutputStreamStream(...)
fs.writeAsync(...).then(function() {
    ...
});
```

Q: What **object** does `writeAsync` return?  
A: Promise[Proto]
Examples

Example 3: Pointer analysis & use analysis

$AudioPrototype$ =

- Volume = function(){};
- Mute = function(){}

function process(button){
  var a = Windows.Media.Audio;
  if(button == 'Mute')
    mute(a);
  if(button == 'Play')
    playSound(a);
  ...
}

function playSound(x){
  x.Volume(30);
  ...
}

function mute(y){
  y.Mute();
}
Pointer analysis

- Uses Datalog declaration and analysis rules
- Accepts an input program represented as a set of relations. Domains:
  - $\mathbf{H}$: Heap-allocated objects and functions
  - $\mathbf{V}$: Program variables
  - $\mathbf{C}$: Call sites
  - $\mathbf{P}$: Properties
  - $\mathbf{Z}$: Integers
- Generate output relations representing the analysis result, e.g.
  - Points-to relation
  - Call graph construction
  - Prototype chain relation
Pointer analysis – inference rules

Output relations

Input relations
Use analysis – Inference rules

- Generate symbolic facts based on the facts and constraints after pointer analysis
- Recover missing flow (arguments, return values and loads) due to missing implementations of libraries.
Abstract locations are used as approximation of runtime object allocation in the program.

Distinguish abstract locations in between $H_A$, $H_L$ and $H_S$

Symbolic locations are introduced for reasoning about abstract locations returned by library calls where flow is dead due to libraries

Reference: “Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries” ppt
Use analysis – Unification

- **Dead flow scenarios**
  - Dead Returns / Dead Arguments / Dead loads

- **Why**
  - Variables within $V$ domain normally have points-to links to heap elements in $H$
  - Ignore library code and use of stubs
  - Missing interprocedural flow in the presence of libraries

- **Solution**
  - Unification strategies
Three unification strategies

- Matching of at least one property
  
  Too many objects get linked

- Matching of all properties
  
  Too few objects get linked
  – Unsoundness & Imprecision

- Prototype-based unification
  1. Disallow commonly-used properties (e.g. prototype, length) for unification
  2. Consider most precise object in the prototype hierarchy to unify first

Best – improve precision
Example – Prototype-based unification

Function `compareIgnoreCase` is defined in application and is used as callback passed into library.

Return arguments `s1` and `s2` have `toLowerCase` property

However, all string constants have this property, should not unify all of them

Consider prototype object: String[Proto]
Iterative Inference Algorithm

- Collects and records occurrence of dead returns/arguments/loads
- Introduces symbol location for each location
- Perform unification: unifying symbolic objects with appropriate application or library abstract locations
- Terminates when no more dead flows can be founded and no more unification can be performed
Use analysis – Other techniques

- Extend Partial Inference to Full Inference
  - Do not assume existence of stubs, fully depends on uses found in applications
  - Allow symbolic location to point to another symbolic location to resolve limited dead loads
    - While in partial inference, symbolic location is only allowed to link to a non-symbolic location

- Namespace Mechanisms
  - Solve the points-to problem of global variable
  - Solve missing prototype problems caused by JavaScript calls to library constructors created by namespace mechanisms.
  - Introduce a special symbolic prototype object to deal with this case

- Array Access and Dynamic Properties
  - Introduce a single symbolic object and inject it into array variables for unification analysis.
Evaluation

- **Experimental Setup**
  - Evaluate both partial and full inference algorithms
  - Evaluation Tool –
    - **Front end**: C#, parses JavaScript application and generates input facts for analysis;
    - **Back end**: Z3 Datalog engine
  - Machine: Windows 7 machine, Xeon 64-bit 4-core CPU, 3.07 GHz with 6 GB of RAM

- Results
Benchmarks

- A set of 25 JavaScript applications

---

**Table 1: Application Sizes**

<table>
<thead>
<tr>
<th>Lines</th>
<th>Functions</th>
<th>Alloc. sites</th>
<th>Call sites</th>
<th>Properties</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>245</td>
<td>11</td>
<td>128</td>
<td>113</td>
<td>231</td>
<td>470</td>
</tr>
<tr>
<td>345</td>
<td>51</td>
<td>282</td>
<td>194</td>
<td>336</td>
<td>1,007</td>
</tr>
<tr>
<td>434</td>
<td>53</td>
<td>369</td>
<td>216</td>
<td>303</td>
<td>1,102</td>
</tr>
<tr>
<td>627</td>
<td>59</td>
<td>341</td>
<td>239</td>
<td>353</td>
<td>1,230</td>
</tr>
<tr>
<td>827</td>
<td>56</td>
<td>381</td>
<td>175</td>
<td>477</td>
<td>1,333</td>
</tr>
<tr>
<td>1,010</td>
<td>70</td>
<td>467</td>
<td>287</td>
<td>354</td>
<td>1,600</td>
</tr>
</tbody>
</table>

**Table 2: Stub Sizes for Widely-Used Libraries**

<table>
<thead>
<tr>
<th>Name</th>
<th>Lines</th>
<th>Functions</th>
<th>Alloc. sites</th>
<th>Call sites</th>
<th>Properties</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Builtin</td>
<td>225</td>
<td>161</td>
<td>1,039</td>
<td>190</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOM</td>
<td>21,881</td>
<td>12,696</td>
<td>44,947</td>
<td>1,326</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WinJS</td>
<td>404</td>
<td>346</td>
<td>1,114</td>
<td>445</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windows 8 API</td>
<td>7,213</td>
<td>2,970</td>
<td>13,989</td>
<td>3,834</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>29,723</td>
<td>16,173</td>
<td>61,089</td>
<td>5,795</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 2:** Approximate stub sizes for widely-used libraries.

**Fig. 9:** Benchmarks, sorted by lines of code.

**Stub size:** 30,000 lines

**Take stubs into account**

**Application size:** 1,587 lines
Call Graph Resolution

- **Baseline**: points-to analysis without considering stubs.
- Partial Inference Algorithm

**Comparison**: baseline resolved much fewer call sites. Partial inference algorithm is effective in recovering missing flow.
Case studies – WinRT API Resolution

- Resolve calls to WinRT APT in Win 8 JavaScript applications
- Partial inference and full inference can find out much more WinRT uses

<table>
<thead>
<tr>
<th>Technique</th>
<th>APIs used</th>
</tr>
</thead>
<tbody>
<tr>
<td>naïve analysis</td>
<td>684</td>
</tr>
<tr>
<td>points-to</td>
<td>800</td>
</tr>
<tr>
<td>points-to + partial</td>
<td>1,304</td>
</tr>
<tr>
<td>points-to + full</td>
<td>1,320</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>4,108</strong></td>
</tr>
</tbody>
</table>
Case studies — Auto-complete

<table>
<thead>
<tr>
<th>Category</th>
<th>Code</th>
<th>Eclipse</th>
<th>IntelliJ</th>
<th>VS 2010</th>
<th>VS 2012</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARTIAL INFERENCE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 DOM Loop</td>
<td>var c = document.getElementById(&quot;canvas&quot;); var ctx = c.getContext(&quot;2d&quot;); var h = c.height; var w = c.w;</td>
<td>x 0</td>
<td>✓ 35</td>
<td>x 26</td>
<td>✓ 1</td>
</tr>
<tr>
<td>2 Callback</td>
<td>var p = {firstName: &quot;John&quot;, lastName: &quot;Doe&quot;}; function compare(p1, p2) { var c = p1.firstName &lt; p2.firstName; if(c != 0) return c; return p1.lastName; }</td>
<td>x 0</td>
<td>✓ 9</td>
<td>x 7</td>
<td>✓ k</td>
</tr>
<tr>
<td>3 Local Storage</td>
<td>var p1 = {firstName: &quot;John&quot;, lastName: &quot;Doe&quot;}; localStorage.setItem(&quot;person&quot;, p1); var p2 = localStorage.getItem(&quot;person&quot;); document.writeln(&quot;Mr.&quot; + p2.lastName + &quot;,&quot; + p2.firstName);</td>
<td>x 0</td>
<td>✓ 50+</td>
<td>x 7</td>
<td>x 7</td>
</tr>
<tr>
<td>FULL INFERENCE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 Namespace</td>
<td>WinJS.Namespace.define(&quot;Game.Audio&quot;, play: function() {}, volume: function() {}); Game.Audio.volume(50); Game.Audio.p;</td>
<td>x 0</td>
<td>✓ 50+</td>
<td>x 1</td>
<td>✓ k</td>
</tr>
<tr>
<td>5 Paths</td>
<td>var d = new Windows.UI.Popups.MessageDialog(); var m = new Windows.UI...</td>
<td>x 0</td>
<td>x 250+</td>
<td>x 7</td>
<td>✓ k</td>
</tr>
</tbody>
</table>

Fig. 15: Auto-complete comparison. * means that inference uses all identifiers in the program. “…” marks the auto-complete point, the point where the developer presses Ctrl+Space or a similar key stroke to trigger auto-completion.
Performance

- Running time of partial and full analysis are quite modest. (full analyses finish under 2-3 seconds)
- Full inference requires more iterations to reach fixpoint (approximately 2 to 3 times as many iterations as partial inference)
- Full inference is fast than partial inference (2 to 4 times faster): cost of stubs
Precision and Soundness

- Manually inspected 20 call sites in 10 benchmarks
- Check if approximated call targets match the actual call targets
  - **OK**: the number of call sites which are both sound and complete
  - **Incomplete**: the number of call sites are sound, but have spurious targets (Imprecision)
  - **Unsound**: the number of call sites for which some call targets are missing
  - **Unknown**: the number of call sites for which unable to determine due to code complexity
  - **Stubs**: the number of call sites which are unsolved due to problematic stubs.

<table>
<thead>
<tr>
<th>App</th>
<th>OK</th>
<th>Incomplete</th>
<th>Unsound</th>
<th>Unknown</th>
<th>Stubs</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>app1</td>
<td>16</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>app2</td>
<td>11</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>app3</td>
<td>12</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>app4</td>
<td>13</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>app5</td>
<td>13</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>app6</td>
<td>15</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>app7</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>app8</td>
<td>12</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>app9</td>
<td>12</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>app10</td>
<td>11</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td><strong>Total</strong>:</td>
<td><strong>135</strong></td>
<td><strong>35</strong></td>
<td><strong>4</strong></td>
<td><strong>5</strong></td>
<td><strong>21</strong></td>
<td><strong>200</strong></td>
</tr>
</tbody>
</table>
Unsoundness & Imprecision

- **Unsoundness**
  - Unable to deal with JSON data being parsed
  - Unable to deal with JavaScript type coercion
    (Type coercion means that when the operands of an operator are different types, one of them will be converted to an "equivalent" value of the other operand’s type. For instance: boolean == integer, the boolean operand will be converted to an integer first)

- **Imprecision**
  - Property names shares between different objects.
  - Stub errors
Conclusions

- Approach proposed combining classic points-to analysis with use analysis
- Able to analyze practical large JavaScript applications using complex windows runtime libraries and sophisticated JavaScript libraries
- Improve precision and scalability
- Useful for other applications: API use discovery and auto-completion
Questions?

Thanks!