CHALLENGES IN POINTER ANALYSIS OF JAVASCRIPT

Ben Livshits

MSR
Area man says: JavaScript leads the pack as most popular programming language.
Two Issues in JavaScript Pointer Analysis
Gulfstream

• JavaScript programs on the web are streaming

• Fully static analysis pointer analysis is not possible, calling for a hybrid approach

• Setting: analyzing pages before they reach the browser
Use analysis

- JavaScript programs interop with a set of reach APIs such as the DOM
- We need to understand these APIs for analysis to be useful
- Setting: analyzing Win8 apps written in JavaScript
Gulfstream

• Staged Static Analysis for Streaming JavaScript Applications, Salvatore Guarnieri, Ben Livshits, WebApps 2009

Abstract

The advent of Web 2.0 has led to the proliferation of client-side code that is typically written in JavaScript. Recently, there has been an upsurge of interest in static analysis of client-side JavaScript for applications such as bug finding and optimization. However, most approaches in static analysis literature assume that the entire program is available to analysis. This, however, is in direct contradiction with the nature of Web 2.0 programs that are essentially being streamed to the user's browser. Users can see data being streamed to pages in the form of page updates, but the same thing can be done with code, essentially delaying the downloading of code until it is needed. In essence, the entire program is never completely available. Interacting with the application causes more code to be sent to the browser.

This paper explores staged static analysis as a way to analyze streaming JavaScript programs. We observe while there is variance in terms of the code that gets sent to the client, much of the code of a typical JavaScript application can be determined statically. As a result, we advocate the use of combined offline-online static analysis as a way to accomplish fast, browser-based client-side online analysis at the expense of a more thorough and costly server-based offline analysis on the static code.

We find that in normal use, where updates to the code are small, we can update static analysis results quickly enough in the browser to be acceptable for everyday use. We demonstrate the staged analysis approach to be advantageous especially in mobile devices, by experimenting on popular applications such as Facebook.

1 Introduction

The advent of Web 2.0 has led to the proliferation of client-side code that is typically written in JavaScript. This code is often combined or mashed-up with other code and content from different third-party servers, making the application only fully available within the user’s browser. Recently, there has been an upsurge of interest in static analysis of client-side JavaScript. However, most approaches in the static analysis literature assume that the entire program is available for analysis. This, however, is in direct contradiction with the nature of Web 2.0 programs that are essentially being streamed to the user’s browser. In essence, the JavaScript application is never available in its entirety; the user interacts with the application, more code is sent to the browser.

A pattern that emerged in our experiments with static analysis to enforce security properties [14] is that while most of the application can be analyzed offline, some parts of it will need to be analyzed on-demand, in the browser. In one of our experiments, while 17% of the JavaScript code is downloaded right away, an additional 62 KB of code is downloaded when visiting event pages, etc. Similarly, Bing Maps downloads most of the code right away; however, requesting traffic requires additional code downloads. Moreover, often the parts of the application that are downloaded later are server-side on the client by referencing a third-party library as a JavaScript URL, common libraries such as jQuery and Prototype.js. Since these libraries change relatively frequently, analyzing this code ahead of time may be inefficient or even impossible.

The dynamic nature of JavaScript, combined with the incremental nature of code downloading in the browser leads to some unique challenges. For instance, consider the piece of HTML in Figure 1. Suppose we want to statically determine what code may be called from the src1.js handler to ensure that none of the invoked functions may block. If we only consider the first SCRIPT block, we will conclude that the src1.js handler may only call function foo. Including the second SCRIPT block adds function bar as a possible function that may be called. Furthermore, if the browser proceeds to download more code, either through more SRC/SCRIPT blocks or JAVASCRIPT objects, more code might need to be consid-
Whole program analysis?
What whole program?
JavaScript programs are streaming
<table>
<thead>
<tr>
<th>Page visited or action performed</th>
<th>Added JavaScript files</th>
<th>KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home page</td>
<td>19</td>
<td>157</td>
</tr>
<tr>
<td>Friends</td>
<td>7</td>
<td>186</td>
</tr>
<tr>
<td>Inbox</td>
<td>1</td>
<td>206</td>
</tr>
<tr>
<td>Profile</td>
<td>1</td>
<td>219</td>
</tr>
</tbody>
</table>

**FACEBOOK FRONT PAGE**

- Home page: 19 files, 157 KB
- Friends: 7 files, 186 KB
- Inbox: 1 file, 206 KB
- Profile: 1 file, 219 KB
## OWA Code Exploration

<table>
<thead>
<tr>
<th>Task</th>
<th>Code</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inbox page</td>
<td>7</td>
<td>1,680</td>
</tr>
<tr>
<td>Expand an email thread</td>
<td>1</td>
<td>95</td>
</tr>
<tr>
<td>Respond to email</td>
<td>2</td>
<td>134</td>
</tr>
<tr>
<td>New meeting request</td>
<td>2</td>
<td>168</td>
</tr>
</tbody>
</table>

![Bar chart showing usage of Outlook Web Access (OWA) tasks]
What does f refer to?
Plan

**Server**
- Pre-compute pointer information offline, for most of the program
- Optionally update server knowledge as more code is observed

**Client**
- When more code is discovered, do analysis of it
- Combine the incremental results with pre-computed results
Is it faster to

1) transfer pre-computed results + add incremental results
2) Compute everything from scratch

Checking a safety property
Simulated Devices

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>CPU coef. $c$</th>
<th>Link type</th>
<th>Latency $L$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>G1</td>
<td>67.0</td>
<td>EDGE</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Palm Pre</td>
<td>36.0</td>
<td>Slow 3G</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>iPhone 3G</td>
<td>36.0</td>
<td>Fast 3G</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>iPhone 3GS 3G</td>
<td>15.0</td>
<td>Slow 3G</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>iPhone 3GS WiFi</td>
<td>15.0</td>
<td>Fast WiFi</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>MacBook Pro 3G</td>
<td>1</td>
<td>Slow 3G</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>MacBook Pro WiFi</td>
<td>1</td>
<td>Slow WiFi</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Netbook</td>
<td>2.0</td>
<td>Fast 3G</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Desktop WiFi</td>
<td>0.8</td>
<td>Slow WiFi</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Desktop T1</td>
<td>0.8</td>
<td>T1</td>
<td></td>
</tr>
</tbody>
</table>
Try Different Configurations

- **Slow devices** benefit from Gulfstream
- A **slow network** can negate the benefits of the staged analysis
- **Large page updates** don’t benefit from Gulfstream

“+” means that staged incremental analysis is advantageous compared to full analysis on the client.
Gulfstream Savings: Fast Devices

10 seconds saved

- profile
- inbox
- friends
- home
Gulfstream Savings: Slow Devices

- Profile
- Inbox
- Friends
- Home
Laptop Running Time Comparison

Break even point:
After 30KB of updates, incremental Gulfstream is no longer faster
Conclusion

- Gulfstream, staged analysis for JavaScript
- WebApps 2010

- Staged analysis
  - Offline on the server
  - Online in the browser

- Wide range of experiments
  - For small updates, Gulfstream is faster
  - Devices with slow CPU benefit most
Pointer Analysis and Use Analysis
Use Analysis

• Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries, Madsen, Livshits, Fanning, in submission, 2013
Motivation:
Win8 App Store

Native C/C++ apps
.NET apps
JavaScript/HTML apps
## Win8 & Web Applications

### Windows 8 App

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

- Call graph discovery
- API surface discovery
- Capability analysis
- Auto-complete
- Concrete type inference
- Runtime optimizations
Practical Applications

- Call graph discovery
- **API surface discovery**
- Capability analysis
- Auto-complete
- Concrete type inference
- Runtime optimizations

Windows.Devices.Sensors
Windows.Devices.Sms
Windows.Media.Capture
Windows.Networking.Sockets
...

26
Practical Applications

- Call graph discovery
- API surface discovery
- Capability analysis
- Auto-complete
- Concrete type inference
- Runtime optimizations

```xml
<Package xmlns="http://schemas.microsoft.com/"
  <Identity Name="51e0e1dc-81a4-4bd0-964a-"
  <Properties>
    <DisplayName>...</DisplayName>
    <Description>...</Description>
  </Properties>
  <Capabilities>
    <Capability Name="videosLibrary" />
    <Capability Name="picturesLibrary" />
    <Capability Name="internetClient" />
    <DeviceCapability Name="webcam" />
  </Capabilities>
</Package>
```
Practical Applications

• Call graph discovery
• API surface discovery
• Capability analysis
• Auto-complete
• Concrete type inference
• Runtime optimizations
Practical Applications

• Call graph discovery
• API surface discovery
• Capability analysis
• Auto-complete
• Concrete type inference
• Runtime optimizations

```javascript
function Node(left, right) {
  this.color = "RED";
  this.height = 0;
  this.left = left;
  this.right = right;
}

var l = new Node(null, null);
var r = new Node(null, null);
var p = new Node(l, r);
```
Practical Applications

- Call graph discovery
- API surface discovery
- Capability analysis
- Auto-complete
- Concrete type inference
- Runtime optimizations

```javascript
function Node(left, right) {
    this.color = "RED";
    this.height = 0;
    this.left = left;
    this.right = right;
}
```

memory layout
Canvas Dilemma

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);

- model querySelector as returning a reference to HTMLElement.prototype

- However, HTMLElement.prototype does not define getContext, so getContext remains unresolved

- Model querySelector as returning any HTML element within underlying page

- Returns elements on which getContext is undefined
Introducing Use Analysis

function handleEvent(event) {
  var elm = document.querySelector("#player video");
  if (event.keyCode == 80) { // User pressed 'p' key
    playVideo(elm);
  }
  if (event.keyCode == 90) { // User pressed 'r' key
    reset(elm);
  }
}

function playVideo(elm) {
  if (elm.muted) {
    elm.volume = 80;
  }
  elm.play();
}

function reset(elm) {
  elm.pause();
  elm.seekable = 0;
}

elm must have: muted and play

elm flows into playVideo

elm must have: pause

elm flows into reset
Pointer vs. Use Analysis

• Pointer analysis deals with “concrete” facts
  • Facts we can observe
    • variables declared in the program
    • allocation sites
• Use analysis deals with the “invisible” part of the heap

• It can exist entirely outside the JavaScript heap

• Constraints flows from callers to callees
driveUtil.uploadFilesAsync(server.imagesFolderId).
    then( function (results) {...} ))

analysis correctly maps `then` to
    WinJS.Promise.prototype.then
Local Storage

```javascript
var json =
    Windows.Storage.
    ApplicationData.current.
    localSettings.values[key];
```

correctly resolves `localSettings` to an instance of `Windows.Storage.ApplicationDataContainer`
## Benchmarks

<table>
<thead>
<tr>
<th>Lines</th>
<th>Functions</th>
<th>Alloc. sites</th>
<th>Call sites</th>
<th>Fields</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>74</td>
<td>606</td>
<td>345</td>
<td>298</td>
<td>1,749</td>
</tr>
<tr>
<td>402</td>
<td>27</td>
<td>236</td>
<td>137</td>
<td>298</td>
<td>769</td>
</tr>
<tr>
<td>434</td>
<td>51</td>
<td>282</td>
<td>224</td>
<td>222</td>
<td>897</td>
</tr>
<tr>
<td>488</td>
<td>53</td>
<td>369</td>
<td>283</td>
<td>1,287</td>
<td></td>
</tr>
</tbody>
</table>

25 Windows 8 Apps: Average 1,587 lines of code
Approx. 30,000 lines of stubs
Evaluation: Summary

• The technique improves call graph resolution

• Unification is both effective and precise

• The technique improves auto-completion compared to what is found in four widely used IDEs

• Analysis completes in a reasonable amount of time
Call Graph Resolution

Median baseline resolution is 71.5%

Median partial resolution is 81.5%
Validating Results

- **Incomplete** is # of call sites which are sound, but have some spurious targets (i.e. imprecision is present)
- **Unsound** is the number of call sites for which some call targets are missing (i.e. the set of targets is too small)
- **Stubs** is the number of call sites which were unresolved due to missing or faulty 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>135</td>
<td>35</td>
<td>4</td>
<td>5</td>
<td>21</td>
<td>200</td>
</tr>
</tbody>
</table>
Auto-complete

• We compared our technique to the auto-complete in four popular IDEs:
  • Eclipse for JavaScript developers
  • IntelliJ IDEA
  • Visual Studio 2010
  • Visual Studio 2012

• In all cases, where libraries were involved, our technique was an improvement
# 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>1  DOM Loop</td>
<td><code>var c = document.getElementById(&quot;canvas&quot;); var ctx = c.getContext(&quot;2d&quot;); var h = c.height; var w = c.width;</code></td>
<td>✗ 0</td>
<td>✓ 35</td>
<td>✗ 26</td>
<td>✓ 1</td>
</tr>
<tr>
<td>2  Callback</td>
<td><code>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; }</code></td>
<td>✗ 0</td>
<td>✓ 9</td>
<td>✗ 7</td>
<td>✓ k</td>
</tr>
<tr>
<td>3  Local Storage</td>
<td><code>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.firstName + &quot;,&quot; + p2.lastName);</code></td>
<td>✗ 0</td>
<td>✓ 50+</td>
<td>✗ 7</td>
<td>✗ 7</td>
</tr>
<tr>
<td>4  Namespace</td>
<td><code>WinJS.Namespace.define(&quot;Game.Audio&quot;, { play: function() {}, volume: function() {} }); Game.Audio.volume(50); Game.Audio.p</code></td>
<td>✗ 0</td>
<td>✓ 50+</td>
<td>✗ 1</td>
<td>✓ k</td>
</tr>
<tr>
<td>5  Paths</td>
<td><code>var d = new Windows.UI.Popups.MessageDialog(); var n = new Windows.UI._</code></td>
<td>✗ 0</td>
<td>✓ 250+</td>
<td>✗ 7</td>
<td>✓ k</td>
</tr>
</tbody>
</table>
Median runtime for partial is 10.5 sec

All benchmarks complete within 22.0 sec

Analysis is not incremental – room for improvement
Two Issues in JavaScript Pointer Analysis

**Gulfstream**
- JavaScript programs on the web are streaming
- Fully static analysis pointer analysis is not possible, calling for a hybrid approach
- Setting: analyzing pages before they reach the browser

**JSCap**
- JavaScript programs interop with a set of reach APIs such as the DOM
- We need to understand these APIs for analysis to be useful
- Setting: analyzing Win8 apps written in JavaScript