Static JavaScript Call Graphs: a Comparative Study

Gábor Antal, Péter Hegedűs, Zoltán Tóth, Rudolf Ferenc, and Tibor Gyimóthy
Introduction

- **JavaScript**
  - increasing popularity for years
  - supports both server and client side
  - code analysis became very important

- **Call graph is a basis for e.g.**
  - **Source Code Analysis and Manipulation**
    - e.g. interprocedural control flow
  - **Software Visualization**

```javascript
function f() {} // node f
function g() { f(); } // edge g->f
```
Call graphs for JavaScript

- Static approaches
  - might miss dynamic calls – e.g. `eval()`
  - are relatively fast
  - require no testbed

- Dynamic approaches
  - find only realized calls
  - are slower
  - require large testbed with high coverage
Motivation

- JavaScript is quite hard to analyze statically because of its dynamic behavior
  - However, several static code analysis tools exist
- We wanted to know
  - is building usable static call graphs feasible?
  - how accurate are the static call graphs?
  - which tools are the most popular?
  - what are the characteristics of these tools?
  - can they handle new EcmaScript standards?
Study design

- Five static JavaScript call graph extraction tools
  - npm callgraph
  - IBM WALA
  - Google Closure Compiler
  - ACG (Approximate Call Graph)
  - TAJS (Type Analyzer for JavaScript)

- Quantitative and qualitative analysis on
  - 26 SunSpider benchmark programs
  - 6 real-world Node.js modules
  - generated inputs
Methodology

Input file

- npm callgraph
  - DOT format
  - Converter (stjscg_convert2json.py)
  - Unified JSON format
  - Graph comparator (stjscg_compare_json.py)
  - Compared JSON

- TAJS
  - DOT format
  - Unified JSON format

- WALA
  - DOT-like format
  - Unified JSON format

- ACG.js
  - DOT-like format
  - Unified JSON format

- Closure compiler
  - Unified JSON format
Results

- SunSpider
- Manually evaluated
Results

- SunSpider
- Manually evaluated
Discussion of results

- npm callgraph (prec.: 91%, recall: 68%)
  - treats calls from anonymous functions as calls from global scope

- Closure Compiler (prec.: 81%, recall: 89%)
  - can find recursive edges
  - relies often only on name matching

- WALA (prec.: 87%, recall: 49%)
  - can detect calls of function arguments
  - can analyze eval() in some cases
Discussion of results

- **TAJS** (prec.: 98%, recall: 71%)
  - can detect calls of function arguments
  - is able to track complex control flows and detect non-trivial call edges

- **ACG** (prec.: 99%, recall: 91%)
  - is able to track complex control flows and detect non-trivial call edges

- **ACG+TAJS** together perform almost perfectly
  - 98% precision, 99% recall

- Combining all tools have 74% precision, 100% recall
Results – Node.js modules

- Only ACG and Closure compiler were able to analyze multifile Node.js modules
- We manually evaluated a statistically significant random sample
  - 716 out of 2281 edges

Manually evaluated sample

<table>
<thead>
<tr>
<th></th>
<th>ACG</th>
<th>Closure</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACG</td>
<td>149/179 (83.2%)</td>
<td>248/297 (83.5%)</td>
</tr>
<tr>
<td>Closure</td>
<td>40/240 (16.7%)</td>
<td></td>
</tr>
</tbody>
</table>

All edges

<table>
<thead>
<tr>
<th></th>
<th>ACG</th>
<th>Closure</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACG</td>
<td>336</td>
<td>1304</td>
</tr>
<tr>
<td>Closure</td>
<td>641</td>
<td></td>
</tr>
</tbody>
</table>
Results – Performance

- Normal PC
- Created both simple and complex inputs
- Ran each analysis 10 times

### Normal PC

<table>
<thead>
<tr>
<th>Category</th>
<th>File</th>
<th>Memory</th>
<th>Runtime</th>
<th>ACG</th>
<th>WALA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>npm callgraph</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Runtime</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Runtime</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Closure Compiler

<table>
<thead>
<tr>
<th>Category</th>
<th>File</th>
<th>Memory</th>
<th>Runtime</th>
<th>TAJS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Closure Compiler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Runtime</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### TAJS

<table>
<thead>
<tr>
<th>Category</th>
<th>File</th>
<th>Memory</th>
<th>Runtime</th>
<th>TAJS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TAJS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Runtime</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

- Small and medium sized input: ACG
- Large input
  - Speed: Closure compiler/TAJS
  - Memory usage: Closure compiler/TAJS
  - Most accurate: ACG
  - npm callgraph and WALA: unusable
- Node.js input: ACG
- Lots of eval(): WALA
- Lots of recursion: Closure compiler
- Most accurate (but slow): ACG + TAJS
Conclusion

- No absolute winner
  - each tool has its strengths and weaknesses
- Combining them carefully might be a good approach
- There are several missed calls
  - In the future, we would like to extend the comparison with dynamic call graphs
- No real ES6 support
- All artefacts are available as online appendix
  - Inputs, call graphs, Venns, our tools, tool patches
New development

- Berkeley and Szeged teams reincarnated ACG
  - Added ES6 support
  - npm package is to be released soon
  - https://github.com/Persper/js-callgraph
- Incorporated into OpenStaticAnalyzer™
  - In next release (soon)
  - http://OpenStaticAnalyzer.github.io
Thank you for your attention!

Gábor Antal

PhD student, Department of Software Engineering
University of Szeged
antal@inf.u-szeged.hu
linkedin.com/in/gantal