# Evaluation and Comparison of Dynamic Call Graph Generators for JavaScript

Zoltán Herczeg and Gábor Lóki
*Department of Software Engineering, University of Szeged, Dugonics tér 13, 6720, Szeged, Hungary*

Keywords: JavaScript, Call graph, Node.js, Comparison, Security, Complexity

Abstract: JavaScript is the most popular programming language these days and it is also the core language of the *node.js* environment. Sharing code is a simple task in this environment and the shared code can be easily reused as building blocks to create new applications. This vibrant and ever growing environment is not perfect though. Due to the large amount of reused code, even simple applications can have a lot of indirect dependencies. Developers may not even be aware of the fact that some of these dependencies could contain malware, since harmful code can be hidden relatively easily due to the dynamic nature of JavaScript. Dynamic software analysis is one way of detecting suspicious activities. Call graphs can reveal the internal workings of an application and they have been used successfully for malware detection. In *node.js*, no tool has been available for directly generating JavaScript call graphs before. In this paper, we are going to introduce three tools that can be used to generate call graphs for further analysis. We show that call graphs contain a significant amount of engine-specific information but filters can be used to reduce such differences.

## 1 INTRODUCTION

The trend of the past years is unbroken, JavaScript is the most popular (StackOverflow, 2018) programming language nowadays. JavaScript has been designed to extend static Web documents with interactive features, but eventually it appeared in other areas, such as server side scripting and embedded systems[1]. Although the syntax of the language is the same on all systems, the application programming interfaces (APIs) provided by each environment can be quite different. From a security perspective, many of these APIs provide access to system resources, such as file systems, network connections, cameras, or private user data, therefore, these resources must be protected from harmful uses and direct attacks.

Detecting malicious behaviour of JavaScript code is among the key areas of protecting users and computer systems. Many solutions (Yu et al., 2007; Guarnieri and Livshits, 2009; Bielova, 2013) that have been developed already are focusing on enforcing security policies. Besides strengthening policies, a more promising approach can be the analysis of JavaScript code in order to detect harmful actions especially on software ecosystems where a large amount of JavaScript code is downloaded and executed from unverified sources. An example for such an ecosystem is the *npm*[2] software registry of *node.js*[3] modules where anybody can upload their own JavaScript modules without any security checks. Injecting a vulnerability, into a potentially transitive and less rigorously tested dependency module, can cause security threats at unexpected points higher in the dependency chain.

A fundamental area of detecting harmful behaviour is analysing the function calls of a software. One form of call information are call graphs and these graphs were used successfully for malware detection on both mobile (Gascon et al., 2013) and non-mobile systems (Elhadi et al., 2012) to detect both known and unknown harmful code.

A call graph (Ryder, 1979) is a directed graph, in which nodes are the functions of a program and edges represent direct function calls between them, and where the direction of an edge points toward the callee. A call graph can be constructed statically, i.e., without executing the program, or dynamically, during the execution. The static analysis of JavaScript programs – helping, among others, call graphs con-

[1] http://jerryscript.net

[2] https://www.npmjs.com

[3] https://nodejs.org



Herczeg, Z. and Lóki, G.
Evaluation and Comparison of Dynamic Call Graph Generators for JavaScript.
DOI: 10.5220/0007752904720479
In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2019)*, pages 472-479
ISBN: 978-989-758-375-9

struction – is a well-researched topic (Jensen et al., 2009; Fink and Dolby, 2012; Madsen et al., 2013; Feldthaus et al., 2013). However, in this paper we are focusing on the less studied dynamic approach to pave the way of dynamic security analysis of JavaScript programs.

Regarding dynamic call graph construction for *node.js*, we raise the following questions:

**RQ1:** Are there any dynamic call graph generator tools for *node.js* modules?

**RQ2:** How do dynamic call graph generators influence the generated graphs of a JavaScript module?

In this paper we investigate the differences of dynamic call graphs constructed by various call graph generators. The rest of the paper is organized as follows. In Section 2, we introduce three essentially different approaches that can generate dynamic call graphs for *node.js* applications and modules. In Section 3, we validate and compare the generated call graphs on a popular JavaScript benchmark, while in Section 4, we compare the call graphs of popular *node.js* modules. In Section 5, we review related work, and finally, conclusions and future works are discussed in Section 6.

# 2 CALL GRAPH TOOLS

Strictly speaking, there are no publicly available tools whose output is a dynamic call graph for *node.js* applications. On the other hand, there are tools which can be extended to construct call information for further processing. In the following sections, we introduce three tools that employ essentially different approaches for generating call graphs.

## 2.1 Jalangi2 Framework

The first tool uses the *Jalangi2* (Sen et al., 2015) framework – an improved version of *Jalangi* (Sen et al., 2013) – which allows dynamic analysis of ECMAScript 5.1 (Ecma International, 2011) code, and supports *node.js* and several web browsers. ECMAScript is the standard which defines the different revisions of JavaScript and its version 5.1 was released in 2011.

Internally, *Jalangi2* uses an instrumentation framework, which can extend an ECMAScript 5.1 source code with event notifications without changing its observable behaviour. Variable assignments, expression evaluation, entering to, or exiting from functions are among the events which can be captured by *Jalangi2*. These events are processed by JavaScript applications called *Jalangi2* analyses. We have created such an analysis (Lóki and Herczeg, 2019) for dynamic call graph construction, which registers handlers for function entry/exit events to collect the nodes and edges of a call graph.

Unlike other tools in this section, *Jalangi2* is a pure JavaScript-based framework that modifies the source code of a JavaScript application, and the modified code is executed by *node.js*.

## 2.2 Nodeprof.js Framework

The second tool is *nodeprof.js* (Sun et al., 2018), which is a dynamic program analysis tool for *node.js* applications. *Nodeprof.js* is built on top of the *Graal-nodejs* project. *Graal-nodejs* is a port of *node.js* which employs the *Graal.js* engine for running JavaScript. The *Graal.js* is an EcmaScript 2017 compatible engine which translates JavaScript source code into a generic abstract syntax tree (AST) representation, and this AST is executed by the *GraalVM* virtual machine.

Similar to *Jalangi2*, *nodeprof.js* also adds event notifications to JavaScript programs. However, instead of instrumenting the source code, the events are linked to the AST representation of the program and modifications required for reporting events are also applied to the AST. To accelerate its adoptation, *nodeprof.js* supports *Jalangi2* analyses for processing events, although the interface is not fully compatible. Fortunately, we were able to reuse our call graph generator analysis (Lóki and Herczeg, 2019) with minor modifications.

Unlike other tools in this section, *nodeprof.js* is a Java-based framework which modifies the *GraalVM* AST representation of a JavaScript source code, and the modified AST is executed by *GraalVM*.

## 2.3 Nodejs-cg: A Modified Node.js

Finally, as the third tool, we have customized *node.js* itself to create a variant called *nodejs-cg* (Lóki and Herczeg, 2019). The default JavaScript engine of *node.js* is the *V8*[4] JavaScript engine, which supports execution tracing. The default behaviour of the built-in tracing is to print information about functions where the execution is entered or exited. We have replaced this tracing mechanism with our own call graph generator.

This call graph generator collects all nodes and edges when a *node.js* application is executed and dumps the whole graph when *node.js* terminates. All

[4]https://developers.google.com/v8

Table 1: Running time and disk space consumed by express.

| | nodejs with tracing enabled | nodejs-cg |
|---|---|---|
| running time | 48 s | 5 s |
| disk space | 161 MB | 0.9 MB |

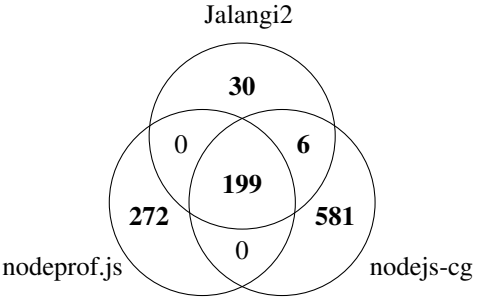


Figure 1: Number of call graph edges on SunSpider.

the information is stored in memory in order to influence the performance of JavaScript execution with printing as little as possible.

Table 1 shows that our approach can be 10 times faster and can take 100 times less space than post-processing the tracing output.

Unlike other tools in this section, the call graph is directly generated by the JavaScript engine of *nodejs-cg*. The source code or the intermediate representation are not modified by the generator tool.

# 3 SUNSPIDER CALL GRAPHS

In this section we compare the call graphs generated from the widely used *SunSpider*[5] benchmark suite in order to examine the basic characteristics of the generated call graphs. Version 1.0.2 of the suite contains 26 programs which are executed one-by-one by a driver application. The source code of *SunSpider* is EcmaScript 5.1 compatible so all three tools (*Jalangi2*, *nodeprof.js*, and *nodejs-cg*) are able to run it.

## 3.1 Node Identification

To compare multiple call graphs of the same program, the same nodes need to be identified in all call graphs. This identification can be done by assigning a unique identifier to each node which is independent from the current execution of the program. Such identifier can be created from the absolute path of the source file where the function is defined and the location of the function start. With this identification mechanism all

[5]https://webkit.org/perf/sunspider/sunspider.html

```
function replaceCallback() {
  return "X";
}
function myReplace(str) {
  return str.replace(/b/g,
      replaceCallback);
}
var str = myReplace('abcba');
```

Figure 2: An example for using the replace() method.

the mentioned call graph generators found the same set of nodes in *SunSpider*.

We have to note that JavaScript supports dynamic script evaluation where scripts are not in files, but are strings constructed at run-time. As a result, they have no path information. Some heuristics can be designed which try to add unique id for these strings, but the identification of an element in such a dynamic code is a complex task. Currently, all of these scripts are assigned to a single *<eval>* node.

## 3.2 Comparison of Found Edges

After we have implemented the same node identification method in each tool, we measured and compared the results. Figure 1 shows the Venn diagram of all call graph edges encountered during the execution of the *SunSpider* benchmark suite. The set of edges found by each generator tool is shown inside a circle corresponding to each tool. The intersections of the circles contain the number of edges which are collected by multiple tools. These edges are called common edges in the following sections. By contrast, unique edges that are found only by a single tool are shown in the non-intersected regions of the circles. If the call graphs generated by the three tools had been the same, only the intersection of the three circles would have contained a non-zero value. However, we found several edges which were found only by one or two tools and the reason for these differences will be discussed in the next subsections.

## 3.3 Jalangi2 and Nodejs-cg

The six edges found by both *Jalangi2* and *nodejs-cg* but not found by *nodeprof.js* are all related to the built-in *replace()* method of JavaScript strings. This method constructs a new string from an existing string where substrings of the original string that matches to a pattern are replaced by another substring. The example shown in Figure 2 uses a regular expression for the pattern and a callback function for generating replacement strings.

A JavaScript engine may implement a built-in function (Ecma International, 2011, section 15) as a

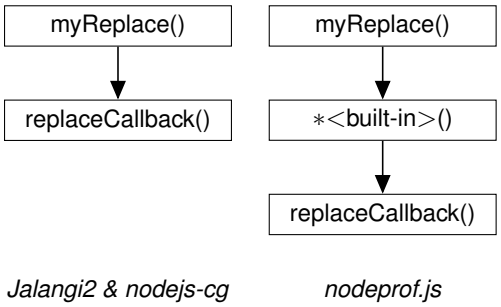


Figure 3: Subgraphs from Figure 2 example.

JavaScript function or as a native function. Native functions are non-JavaScript functions which can be called from JavaScript. They are usually compiled as part of the JavaScript engine and stored in the same binary executable file. JavaScript engines may implement the *replace()* function as a native function, and they often cannot track the entry/exit of such functions.

Figure 3 shows two different subgraphs which connect the nodes of the *myReplace()* and *replaceCallback()* function declared in Figure 2. The subgraph on the left is part of the call graphs generated by *Jalangi2* and *nodejs-cg*. In this case, there is no node assigned to the *replace()* method, because *Jalangi2* cannot instrument any built-in functions, and *nodejs-cg* cannot detect native function calls. Hence, when the *replace()* method calls its callback function, an edge is created which directly connects the *myReplace()* and *replaceCallback()* functions.

Unlike the other two tools, *nodeprof.js* captures the built-in function calls, although it assigns the same node called ∗<*built-in*>*()* to all of them, which makes impossible to tell which particular built-in function is called. The subgraph including the ∗<*built-in*>*()* node is visible on the right side of Figure 3.

## 3.4 Unique Jalangi2 Edges

Twenty-seven edges – out of the thirty edges found only by *Jalangi2* in Figure 1 – are related to module loading. In *node.js*, every JavaScript input file is loaded as a module. First, the source code loaded from a file is wrapped into a new JavaScript function as shown in Figure 4. The wrapped source code is then evaluated by the JavaScript engine of *node.js* which returns with a JavaScript function if no syntax error is found. Then the returned function is called from the module loader with the appropriate arguments. This explains why we consider *node.js* modules as functions: because their source code is wrapped into a JavaScript function first.

```
console.log('a.js loaded');
```

(a) Original source code

```
(function(exports, ...) {
console.log('a.js loaded');
})
```

(b) Wrapped source code

Figure 4: Example for source code wrapping.

```
function f() { eval("") }
Object.prototype.g = function() {}
f();
```

Figure 5: Invalid edges by Jalangi2.

The module loading in *node.js* is mostly performed by JavaScript helper functions. This process is clearly visible on the call graphs generated by *nodeprof.js* and *nodejs-cg*. However, *Jalangi2* does not track the internal JavaScript function calls of *node.js*, and it adds a direct edge between two modules if one of them loads the other. Since the *SunSpider* driver loads twenty-six programs, it is represented by the same amount of edges between the driver and the loaded programs. The driver itself is loaded as a module and the edge between a virtual entry node and the driver is the twenty-seventh edge.

The *string-tagcloud.js SunSpider* program performs a sort operation on an *Array* object. This operation is done by calling the *sort()* built-in method, which has a callback argument similar to the *replace()* method. Since *Jalangi2* only detects the calling of the callback function, a direct edge is added between the caller of *sort()* method and the callback function. As for *nodejs-cg*, the *sort()* method is implemented in JavaScript and the appropriate nodes and edges are added to the call graph. *nodeprof.js* assigns the ∗<*built-in*>*()* node for the *sort()* method similar to *replace()*.

The last two unique *Jalangi2* edges out of the thirty are invalid. Figure 5 shows a simplified example for creating such invalid edges. A new function property called *g* is assigned to the *Object.prototype* built-in object. This function should never be called by the *f* function according to the ECMAScript standard but it does happen when this source code is executed in a *Jalangi2* environment. *Jalangi2* captures the calling of the built-in *eval* function which allows dynamic evaluation of JavaScript source code and replaces the source code passed to the *eval* function with an instrumented one. During the instrumentation, the properties of an object, which inherits the properties of *Object.prototype*, are enumerated and these properties are called as functions. Since the instrumented *g* function is called as well, the generator adds a new edge between *f* and *g* functions. Simi-

Table 2: Call graph edge groups by nodeprof.js.

| group name | number of edges | |
|---|---|---|
| common | 199 | (42.3%) |
| node.js init | 107 | (22.7%) |
| JS built-ins | 99 | (21.0%) |
| module loading | 66 | (14.0%) |
| total | 471 | (100.0%) |

Table 3: Call graph edge groups by nodejs-cg.

| group name | number of edges | |
|---|---|---|
| common | 199 | (25.3%) |
| common Jalangi2 | 6 | (0.8%) |
| nodejs init | 491 | (62.5%) |
| JS built-ins | 22 | (2.7%) |
| module loading | 68 | (8.7%) |
| total | 786 | (100.0%) |

lar invalid edges appear in the actual call graph after the *string-tagcloud.js* program adds a function property to *Object.prototype*. This issue shows the implementation challenge of JavaScript-based instrumentation tools: they need to avoid interacting with the JavaScript environment.

### 3.5 Nodeprof.js and Nodejs-cg Edges

Edges of the call graph generated by *nodeprof.js* can be divided into four groups. Table 2 shows these groups and the number of edges for each group. The value of the *common* group is the same as the value inside the intersection of the three circles in Figure 1. Although this group is the largest, only less than half of the edges belong here. The remaining three groups contain those 272 edges which were found only by *nodeprof.js*.

The edges of the *node.js init* group are recorded during *node.js* initialization. So, they are part of every call graph generated by *nodeprof.js* regardless of the application run by *node.js*. The *JS built-ins* group contains those edges that involve internal functions provided by the JavaScript interpreter, e.g., calling JavaScript built-ins or calling other functions by these built-ins. We should emphasize that *node.js* specific helpers are not part of this group, they belong to the last group, which is called module loading. *SunSpider* does not use any features of *node.js* except the driver program, which loads the other programs as modules. Hence, the edges in this group are all related to *module loading*.

To put the values of Table 2 in context, a similar grouping is shown for *nodejs-cg* in Table 3. The only difference is an extra group called *common Jalangi2*, which contains those six edges that are found only by *Jalangi2* and *nodejs-cg*. These six edges have been discussed before. Compared to *nodeprof.js*, the number of edges in *node.js init* group is 4.5 times bigger and this difference has two reasons. First, the JavaScript engine of *node.js* has its own initialization process, where several scripts written in JavaScript are executed. The other reason is that *nodeprof.js* does not record anything before an analysis is loaded and *node.js* is partly initialized by this time. The number of *JS built-ins* edges are lower than in *nodeprof.js*, since several built-in functions are implemented as native functions in *node.js*. The edges representing these native function calls are not part of the call graph. The *module loading* group has the lowest difference between *nodejs-cg* and *nodeprof.js* because this group depends only on *node.js* internals. The reason for this slight change is that the tools use a different version of *node.js*.

Although the number of edges in the *common* group is the same for *nodejs-cg* and *nodeprof.js*, only twenty-five percent of the edges belong here from the call graph generated by *nodejs-cg*. This ratio is even lower than *nodeprof.js*, where it was around forty percent. The rest of the edges are engine specific and they reveal information about the internal workings of an engine rather than how *SunSpider* works.

## 4 CALL GRAPHS OF APPLICATIONS

In the previous section, we compared the edges of multiple call graphs generated from the *SunSpider* benchmark suite. We found that some call graphs have a large amount of unique edges, e.g., seventy-five percent of the edges are unique in the call graph generated by *nodejs-cg*. However, *SunSpider* is a relatively small benchmark suite, so it would be beneficial if additional investigation was done with other applications before drawing conclusions.

In this section, we compare the call graphs generated from seven popular *node.js* modules (Gyimesi et al., 2019) from GitHub. Each module has a testing system which runs on top of *node.js* and spawns other *node.js* instances to do the testing. The previously discussed call graph generators can run as these instances since they are drop-in replacements for a *node.js* executable. The only difference is that besides running tests, the generators also produce call graphs. After the testing is completed, a final call graph, which is the union of the produced call graphs, is created. The final call graph contains all function calls that happened during testing , including the function calls of test drivers, test cases, engine internals,

Table 4: Number of call graph edges found by nodeprof.js and nodejs-cg.

| | All call graph edges | | | Non-builtin call graph edges | | | Module call graph edges | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | nodeprof.js | common | nodejs-cg | nodeprof.js | common | nodejs-cg | nodeprof.js | common | nodejs-cg |
| bower | 11821 | 10273 | 10422 | 8063 | 11967 | 8545 | 133 | 18455 | 82 |
| doctrine | 2778 | 2333 | 2692 | 1644 | 2786 | 2053 | 17 | 3567 | 6 |
| express | 6633 | 8413 | 6028 | 3842 | 9990 | 4306 | 6 | 11452 | 6 |
| hessian | 3344 | 2064 | 3206 | 2205 | 2518 | 2667 | 5 | 3441 | 6 |
| jshint | 2979 | 1957 | 2763 | 1945 | 2249 | 2373 | 5 | 3184 | 6 |
| pencilblue | 7779 | 5579 | 7089 | 5123 | 6852 | 5659 | 16 | 9989 | 24 |
| request | 8359 | 3639 | 7517 | 5875 | 4674 | 6316 | 22 | 7607 | 24 |

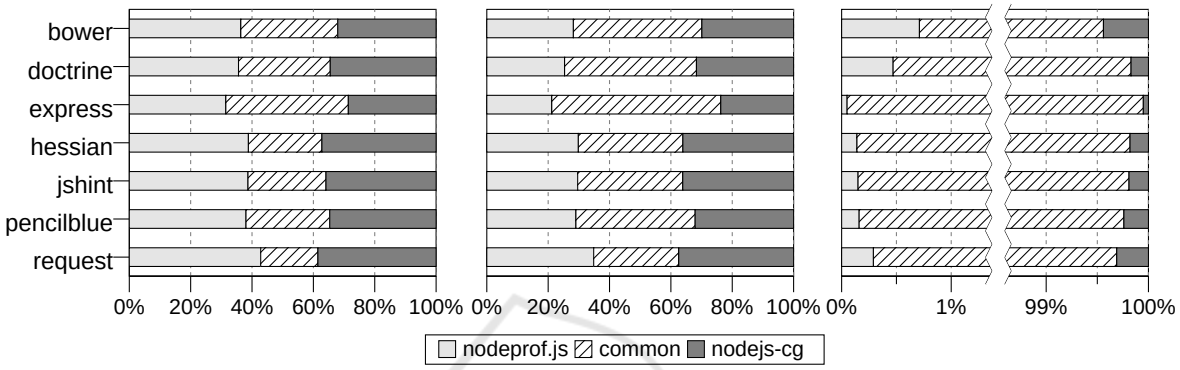


Figure 6: Distribution of call graph edges found by nodeprof.js and nodejs-cg.

and helper modules. Checking helper modules is important from a security perspective, since any of them may contain malware. In this section, we focus on *nodeprof.js* and *nodejs-cg* only, and exclude *Jalangi2* because all applications require ECMAScript 2015 support for testing but *Jalangi2* is ECMAScript 5.1 compatible only.

## 4.1 Overview of the Comparison Results

Table 4 shows three subtables separated by vertical lines, and Figure 6 shows the visual representation of these tables in the same order. Each line in these tables contains three numbers that represent the number of call graph edges recorded by *nodeprof.js* and *nodejs-cg* during the testing of a module. The first and last numbers show the unique edges collected by *nodeprof.js* and *nodejs-cg* respectively, and the *common* column shows those edges that were found by both tools. The name of the module whose call graph information is presented in a given line is shown in the leftmost column before the first subtable.

The first subtable in Table 4 shows the number of all edges which are captured by *nodeprof.js* and *nodejs-cg*. This table reveals that there are considerable differences between the call graphs. For example, the number of unique edges for the *request* program is more than twice as big as the common edges. The large number of unique edges makes the comparison difficult. To simplify the comparison, we applied run-time filters to eliminate certain differences observed during the comparison of *SunSpider* call graphs.

The first filter ignores the built-in functions of the JavaScript interpreter. The effect of this filter is similar to inlining the code of a built-in function into its caller function. Subgraphs – similar to the one above at the *nodeprof.js* label in Figure 3 – are simplified to the one above the *jalangi2 & nodejs.cg* label. The results after applying this filter are shown in the middle subtable of Table 4. Although the number of unique edges have decreased, their number is still high.

The previous filter was extended to ignore those *node.js* core modules which are embedded into the *node.js* executable as well. The source codes of these modules are different since the two tools use different versions of *node.js*. The rightmost subtable in Table 4 shows the edges that remained after applying the new filter. Since the number of unique edges is quite low in this table, we can conclude that the majority of differences are caused by *node.js* and JavaScript engine internal functions.

## 4.2 Remaining Differences

The unique edges at the rightmost table in Table 4 were checked by hand, and they fall into one of the following two categories.

The first category is related to the *class* language construct which supports explicit and implicit constructors. When a class is instantiated, *nodejs-cg* captures explicit constructor calls only. However, *node-prof.js* captures both constructor calls, and the first character of the *class* keyword is identified as the location of constructors. Since the location of a function is part of the unique identifier, the nodes assigned to constructors cannot be paired in the two call graphs.

The second category is test result differences. Because the versions of *node.js* used by the two tools are different, a few tests have run-time issues. E.g., one test may fail with one version of the execution environment but pass with the other. Obviously, the function calls of a test after a fail are missing from the corresponding call graph, but new function calls are often added as well by the error recovery system of the testing environment. Furthermore, some tests are skipped on certain versions of *node.js*.

### 4.3 Comparison Findings

We found that a large percentage of call graph edges may be engine specific. Applications performing high level analysis of JavaScript software should be aware of this fact and may want to adjust their analysis strategy accordingly. Call graph generators may also help improving certain analyses by filtering unwanted edges. For example, filtering out engine specific nodes simplifies the comparison of call graphs, which can be useful for finding the same malware even if the call graph is produced by a newer version of *node.js* extended with the same call graph generator.

## 5 RELATED WORKS

Call graphs can be constructed statically or dynamically. There are several tools for generating static call graphs from JavaScript code (Jensen et al., 2009; Fink and Dolby, 2012; Madsen et al., 2013; Feldthaus et al., 2013). Their strength is that they can process JavaScript code regardless of the execution environment (web browser, *node.js*, etc.). Their weakness is their precision. JavaScript is a highly dynamic language where all functions are objects constructed from a source code and a lexical environment, and it is difficult to predict which particular function object is used by a function call. Some tools try to improve these call graphs when well-known APIs are used. For example, the formalized event emitter API of *node.js* can be used to register listener functions, and these calls can be detected by a static, event-based call graphs analysis (Madsen et al., 2015).

Dynamic call graph generators for web browsers are available (Toma and Islam, 2014; Nguyen et al., 2014). The aim of these tools is to provide run-time feedback to integrated development environments (IDEs) and ultimately to developers about the execution of their own code in a web browser. Unlike our tools, these tools do not support *node.js*.

## 6 SUMMARY

In this paper, we have introduced three new call graph generator tools which are based on *Jalangi2*, *node-prof.js*, and *node.js*, and are developed to fill the missing dynamic call graph gap in the field of *node.js* applications. In their original form, none of them were capable of generating call graphs. We have shown that these tools can be extended to extract call graph information. Hence, the answer to the first research question (**RQ1**) – if there are any dynamic call graph generator tools for JavaScript modules – is that there are such tools available now (Lóki and Herczeg, 2019).

To answer the second research question (**RQ2**) – how dynamic call graph generators influence the generated graphs of a JavaScript module – we have compared the call graphs generated by our three tools and noticed that the call graphs of our test programs have many unique edges. These edges are related to the different handling of built-in functions, ECMAScript *classes*, and different versions of *node.js*. Furthermore, we found some invalid edges which has revealed an issue in the instrumentation framework of *Jalangi2*. Thus, our conclusion is that the internal operation of dynamic call graph generators can significantly influence generated call graphs. However, we also presented that these differences can be eliminated by proper filtering mechanisms that may improve the efficiency of further dynamic analyses.

Our future plan is to extend this research by generating more detailed call information. We aim at investigating call paths (call chains). We also plan to analyse how such information can be utilized to detect unusual program activity.

## ACKNOWLEDGMENTS

This research was partially supported by the Hungarian Government and the European Regional Development Fund grant GINOP-2.3.2-15-2016-00037 (“Internet of Living Things”) and by the Min-

istry of Human Capacities, Hungary grant 20391-3/2018/FEKUSTRAT.

## REFERENCES


Bielova, N. (2013). Survey on JavaScript security policies and their enforcement mechanisms in a web browser. *The Journal of Logic and Algebraic Programming*, 82(8):243 – 262.

Ecma International (2011). Ecmascript language specification 5.1 edition. http://www.ecma-international.org/ecma-262/5.1.

Elhadi, A., Maarof, M., and Hamza Osman, A. (2012). Malware detection based on hybrid signature behaviour application programming interface call graph. *American Journal of Applied Sciences*, 9:283–288.

Feldthaus, A., Schäfer, M., Sridharan, M., Dolby, J., and Tip, F. (2013). Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 752–761. IEEE Press.

Fink, S. and Dolby, J. (2012). WALA-The TJ Watson Libraries for Analysis. http://wala.sourceforge.net.

Gascon, H., Yamaguchi, F., Arp, D., and Rieck, K. (2013). Structural detection of Android malware using embedded call graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, pages 45–54. ACM.

Guarnieri, S. and Livshits, V. B. (2009). Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, volume 10, pages 78–85.

Gyimesi, P., Vancsics, B., Stocco, A., Mazinanian, D., Beszédes, Á., Ferenc, R., and Mesbah, A. (2019). BugsJS: a benchmark of JavaScript bugs. *12th IEEE International Conference on Software Testing, Verification and Validation*. https://github.com/bugsjs.

Jensen, S. H., Møller, A., and Thiemann, P. (2009). Type Analysis for JavaScript. In *International Static Analysis Symposium*, pages 238–255. Springer.

Lóki, G. and Herczeg, Z. (2019). Dynamic call graph generators for JavaScript. https://github.com/szeged/js-call-graphs/tree/call-graphs.

Madsen, M., Livshits, B., and Fanning, M. (2013). Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 499–509. ACM.

Madsen, M., Tip, F., and Lhoták, O. (2015). Static analysis of event-driven node.js JavaScript applications. *SIGPLAN Not.*, 50(10):505–519.

Nguyen, H. V., Kästner, C., and Nguyen, T. N. (2014). Building call graphs for embedded client-side code in dynamic web applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 518–529. ACM.

Ryder, B. (1979). Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, 5:216–226.

Sen, K., Kalasapur, S., Brutch, T., and Gibbs, S. (2013). Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498. ACM.

Sen, K., Sridharan, M., and Adamsen, C. Q. (2015). Jalangi2 dynamic analyses framework for JavaScript. https://github.com/Samsung/jalangi2.

StackOverflow (2018). Stack Overflow annual developer survey. https://insights.stackoverflow.com/survey/2018.

Sun, H., Bonetta, D., Humer, C., and Binder, W. (2018). Efficient dynamic analysis for node.js. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 196–206. ACM.

Toma, T. R. and Islam, M. S. (2014). An efficient mechanism of generating call graph for JavaScript using dynamic analysis in web application. In *2014 International Conference on Informatics, Electronics Vision*, pages 1–6.

Yu, D., Chander, A., Islam, N., and Serikov, I. (2007). JavaScript instrumentation for browser security. *SIGPLAN Not.*, 42(1):237–249.