IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 3 February 2023, accepted 28 February 2023, date of publication 10 March 2023, date of current version 16 March 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3255984

RESEARCH ARTICLE

Is JavaScript Call Graph Extraction Solved Yet? A Comparative Study of Static and Dynamic Tools

GÁBOR ANTAL, PÉTER HEGEDŰS, ZOLTÁN HERCZEG, GÁBOR LÓKI, AND RUDOLF FERENC
Department of Software Engineering, University of Szeged, 6720 Szeged, Hungary

Corresponding author: Péter Hegedűs (hpeter@inf.u-szeged.hu)

This work was supported in part by the European Union Project within the framework of the Artificial Intelligence National Laboratory under Grant RRF-2.3.1-21-2022-00004; in part by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA Funding Scheme, under Project TKP2021-NVA-09; and in part by the University of Szeged Open Access Fund under Grant 5913. The work of Péter Hegedűs was supported by the Bolyai János Scholarship of the Hungarian Academy of Sciences.

ABSTRACT Code analysis is more important than ever because JavaScript is increasingly popular and actively used, both on the client and server sides. Most algorithms for analyzing vulnerabilities, finding coding issues, or inferring type depend on the call graph representation of the underlying program. Luckily, there are quite a few tools to get this job done already. However, their performance in vitro and especially in vivo has not yet been extensively compared and evaluated. In this paper, we compare several approaches for building JavaScript call graphs, namely five static and two dynamic approaches on 26 WebKit SunSpider programs, and two static and two dynamic approaches on 12 real-world Node.js programs. The tools under examination using static techniques were *npm call graph*, *IBM WALA*, *Google Closure Compiler*, *Approximate Call Graph*, and *Type Analyzer for JavaScript*. We performed dynamic analyzes relying on the *nodejs-cg* tool (a customized Node.js runtime) and the *NodeProf instrumentation and profiling framework*. We provide a quantitative evaluation of the results, and a result quality analysis based on 941 manually validated call edges. On the SunSpider programs, which do not take any inputs, so dynamic extraction could be complete, all the static tools also performed well. For example, TAJS found 93% of all edges while having a 97% precision compared to the precise dynamic call graph. When it comes to real-world Node.js modules, our evaluation shows that static tools struggle with parsing the code and fail to detect a significant amount of call edges that dynamic approaches can capture. Nonetheless, a significant number of edges not detected by dynamic approaches are also reported. Among these, however, there are also edges that are real, but for some reason the unit tests did not execute the branches in which these calls were included.

INDEX TERMS Call graph, comparative study, dynamic code analysis, JavaScript, static code analysis.

I. INTRODUCTION

JavaScript is the most popular programming language (and has been since 2014), according to GitHub statistics [15] (followed by Python and Java). The TIOBE Index chooses the fastest-growing programming language each year and honors it with the ''Programming Language of the Year'' title. This award was given to JavaScript in 2014, and since then JavaScript remained in the top 10 languages. JavaScript might be a reasonable choice because it can be used both on the server and client side, and has really large library support. Because of its growing popularity, many projects now utilize JavaScript as their main programming language for both server and client-side modules. As a consequence, code analysis of JavaScript programs has also become a major topic. Numerous code analysis techniques rely on the program's call graph representation. A call graph contains nodes (that represent program functions) and edges (that connect nodes if there is at least one function call between the respective functions). Various quality and security flaws can be detected using this program representation, for example, it can be used to detect functions that are never called or as a visual representation that aids in understanding the code easier. Call graphs can be used to determine whether the correct number of parameters

The associate editor coordinating the review of this manuscript and approving it for publication was Michael Lyu.

are passed to function calls or as a starting point for further analysis, such as a full interprocedural control flow graph (ICFG). Various type analysis algorithms can be performed using control flow graphs [36], [42], [47], [53]. This program representation is also beneficial in other fields of research, such as mutation testing [50], automated refactoring [35], or defect prediction [25].

As call graphs are such a crucial data structure, their precision determines the precision of the code analysis algorithms that rely on them. Creating precise call graphs for JavaScript, an inherently dynamic, type-free, and asynchronous language, is a challenging task. On the one hand, static approaches have the evident disadvantage of missing dynamic call edges from non-trivial *eval*(), *bind*(), or *apply*() usages (i.e., reflection). Furthermore, they could be overly conservative, recognizing edges that, while statically valid, are never realized for any input in practice. They are, however, faster and more efficient than dynamic analysis techniques and do not require an extensive testbed for the analyzed program. Dynamic approaches, on the other hand, identify only real call edges, but the completeness of their results is highly dependent on the quality of the underlying program's test cases. Therefore we need to learn more about the state-of-the-art static and dynamic JavaScript call graph building techniques to better understand their capabilities and limitations in comparison to each other (both in terms of tools and approaches).

This paper, which is an extension of our previously presented conference paper [21], compares two dynamic analysis-based call graph extraction approaches to several well-known and popular static analysis-based call graph extraction approaches [48], to answer the following research questions:

- **RQ1**: Are all static call graph extractor tools find the same call edges or there are code constructs that one can handle but not the other?
- **RQ2**: Do the static call graph extractor tools find all the real edges detected by dynamic analysis?
- **RQ3**: Are static call graph extractor tools able to detect true call edges in real-world programs that the dynamic analysis misses?

To answer these questions, we evaluate five completely different static analysis-based tools: TAJS (Type Analyzer for JavaScript) [14], ACG (Approximate Call Graph) [1], Google Closure Compiler [4], IBM WALA [6], and npm callgraph [10]; and two dynamic tools: NodeProf [9] and nodejs-cg [8] quantitatively, to determine the different calls each tool can detect and how results of static analysis-based tools relate to the dynamic analysis-based results. We also perform a quality analysis of the results, which means comparing and validating the call edges found and analyzing the differences. In addition, we compare the results of the static and dynamic tools to get a sense of how precise static analysis is overall.

To perform our analyses, we needed inputs, however, there is no existing community-accepted benchmark for evaluating JavaScript call graph builder algorithms. To overcome this problem, we identified two different sets of inputs: first, simple, one-file inputs (in this case, we used the SunSpider benchmark), and second, multi-file real projects (we chose several popular Node.js modules).

Regarding the results of the SunSpider analysis, we discovered variations in the numbers, precision, and types of call edges that different tools report. However, there were significant intersections between the reported edges. We concluded that TAJS has the highest precision based on a manual evaluation of 348 call edges, with more than 97% of the edges it found being true positives. The union of all true edges found by the five tools revealed that ACG and TAJS had the highest recall (93%). Nevertheless, Closure detected true positive edges that all other static tools had missed. TAJS obtained an accuracy of 97% but failed to detect any unique edges (edges that other static tools miss). The call graph built by TAJS was also the one that was most similar to that of the dynamic tool. Additionally, we examined the combinations of static tools and observed that none of them could find all true edges while the combinations introduce a lot of false ones; the combined precision was only 53%. As for the similarity between the static call graphs and the dynamically constructed ones, our results varied to a great extent. One issue we found was that many of the missing dynamic edges were not realized in any runs since the test inputs were not complete. The dynamic nature of JavaScript, on the other hand, prevented static techniques from reliably identifying edges.

However, as far as the analysis results of the Node.js modules are concerned, we also found a large variance in the results. Since none of the tools other than ACG were capable of analyzing multi-file projects, ACG was the only tool we could use in addition to the dynamic approaches. However, we were able to use ACG's two call graph building strategies. We found that while the dynamic tools have perfect precision, the highest precision of ACG was only 34.20%. Nevertheless, the recall values of static and dynamic approaches are surprisingly not that different, ranging from 58.40-69.52%. As for the combination of the two approaches, we achieved a perfect recall with a precision of 39.49% (at most).

To summarize, the main contributions of this work (not included in our previous paper [21]) are:

- The quantitative and results quality analysis of the static and dynamic tools on 26 SunSpider benchmark programs.
- The evaluation and comparison of ACG (the only feasible static tool) and the dynamic approaches on 12 widely used Node.js modules.
- A manually validated dataset of call edges found by these tools, which is publicly available in an online appendix[1] (it contains all the tool modification patches as well).

[1] https://doi.org/10.5281/zenodo.7104954

Nevertheless, we would like to clarify that our paper is a comparative study on the creation of JavaScript call graphs, and it was not our intention to create a full-blown benchmark on this topic.

The rest of the paper is organized as follows. The methodology we used for tool selection and comparison is described in III. In Section IV, we present the findings of our quantitative and result quality analyses. We list the possible threats to the validity of our results in Section V. We conclude in Section VI.

II. RELATED WORK

Call graphs have long been present in the field of program analysis, so they can be considered a mature technique. The first publications mentioning call graphs were published in the 1970s [33], [38]. For example, malware classification can be built upon call graph clusters [44], software faults can be detected by using call graphs [30], not to how important call graphs are in the field of debugging [54]. Based on their construction method, we can divide call graphs into two basic subgroups, they can be either dynamic [60] or static [51]. We can combine dynamic and static call graphs to construct hybrid call graphs [22], [55].

Running the application and gathering runtime information about the interprocedural flow results in **dynamic call graphs** [31]. Techniques such as source code instrumentation can be used to generate dynamic call graphs [29].

There is no need to execute the program in the case of **static call graphs**. Static call graphs are constructed by static analysis of a program's source code. These type of call graphs frequently contains non-realizable edges. Even if the source code cannot be run, static call graphs can be generated – in most cases. The combination of static and dynamic analysis techniques, i.e. hybrid solutions guarantee more precise call graphs and, as a result, more precise analyses [32].

The popularity of scripting languages such as JavaScript and Python has increased the need for program analysis in these languages [36]. Constructing precise static call graphs for dynamic scripting languages, on the other hand, is a difficult task that has yet to be entirely solved. Reflective use of the interpreter (when any string can be interpreted as a source code, for example, *eval()* in JavaScript, *exec()* in Python), or any other dynamic binding (e.g.: *apply()*, and *bind()*) construction of the languages make static code analysis extremely difficult. There are various techniques for creating such static call graphs in JavaScript, with varying levels of effectiveness [26], [36], [37]. However, call graphs generated in this manner are frequently limited, and none of the studies cover newer ECMAScript standards (ECMAScript 6 and newer) completely.

Wei and Ryder proposed blended taint analysis for JavaScript, which employs the combination of static and dynamic analysis approaches [59]. By applying dynamic analysis, they could collect information for even those situations that are hard to analyze statically. Dynamic results (execution traces) are propagated to a static infrastructure which embeds a call graph builder as well. This call graph builder module makes use of the dynamically identified calls. However, in the case of pure static analysis, they wrapped the WALA tool to construct a static call graph. In our study, we also included WALA.

Feldthaus et al. proposed an approximation approach for building a call graph [36] that ensured scalable JavaScript IDE support. Madsen et al. focused on the issues caused by the project's included libraries [47]. To enhance scalability and precision, they employed pointer analysis and a novel ''use analysis''. Dijkstra did a thorough evaluation of various static JavaScript call graph building algorithms in his thesis [28]. His research is similar to our comparative, however, he focused on evaluating the various conceptual algorithms and he did the implementation himself, in Rascal. Furthermore, since 2014, a lot has happened in this field. In contrast, we focus on the comparison of mature and state-of-the-art tool implementations on these algorithms that are suitable for in-practice use.

In their work, Salis et al. also performed a comparison in the field of generating static call graphs for Python [56]. They also proposed their own tool which they used in the comparison. We specifically deal with call graphs for JavaScript in this work.

There are further studies with the purpose of creating frameworks for comparing call graph construction algorithms [20], [46]. Nevertheless, these frameworks are for call graphs built from Java or C code. Call graphs are frequently used for preliminary analysis to determine whether or not the code may be optimized. Unfortunately, because they are specific to Java and C, we were unable to use these frameworks as-is for comparing JavaScript call graphs.

III. METHODOLOGY

In this section, we present the methodology used in the research. We adhered to Sim et al.'s guidelines [57], however, our primary intention was not to create a complete benchmark, rather than a comparative study on the state-of-the-art JavaScript call graph extraction tools (that of course required us to assemble a proto-benchmark). Our methodology consists of five high-level steps.

1) Selection of the subjects systems
2) Input selection
3) Preparation of the tools
4) Execution of the analyses
5) Comparison and result evaluation.

In the following paragraphs, we describe the context of the designed study, then give a deeper technical description of the execution details.

Definition. First of all, the term *call graph* might not be precise in itself, so we precisely defined what call graph means in our study. In this paper, we work with call graphs, where:

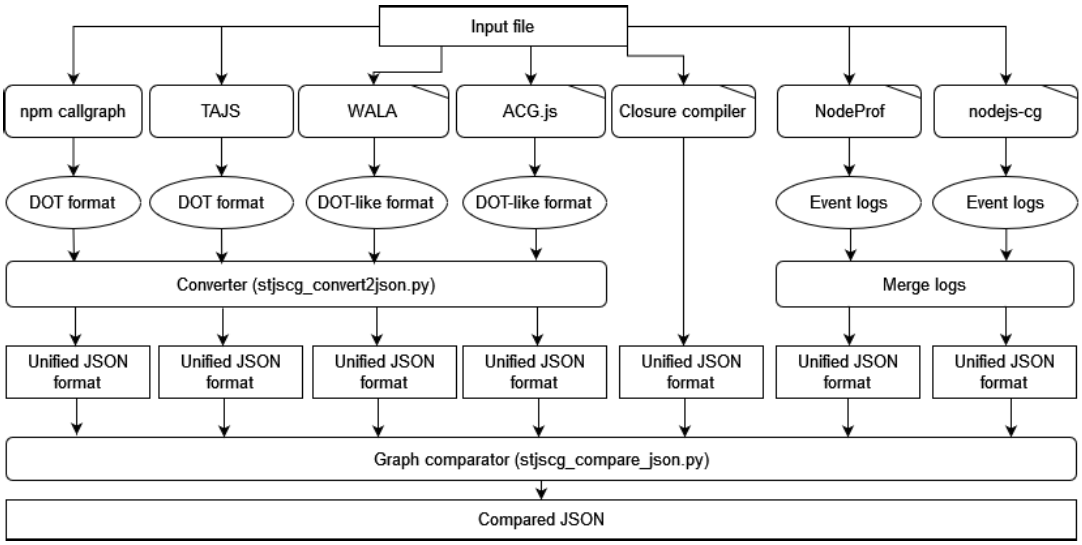

FIGURE 1. Methodology overview.

- The nodes represent program functions (functions are identified by the name of the containing file, and the exact source code position (line and column) where the function starts),
- A directed edge connects two nodes and represents a call from one function to another (i.e., function *a()* calls function *b()*),
- Because there may be just one or zero edges between two nodes, we track only if a call from one function to another is feasible, but we ignore its multiplicity (i.e., we do not count how many call sites a call may happen). This is because not all tools can detect multiple calls and, in any case, we intended to keep to the most basic definition of call graphs.

At first sight, it may seem reasonable to compare graphs in our study in the "traditional" way. Nonetheless, our first intuition was that the tools might miss particular edges, which would have made comparing graphs much more difficult. Instead, in this paper, we will study the set of edges that compose graphs. We would like to mention that our call graph outputs can be used at any time to construct a graph on which any graph theory algorithm can be executed.

Scope. As we are dealing with call graphs in our study, we needed call graph extraction tools. However, selecting a suitable set of such tools can be tricky, a simple search might result in hundreds of potential tools. Hence, we have formalized the tool selection criteria and we took into account tools that met the following criteria: i) can produce a function call graph from a JavaScript program, ii) are open-source and free, and iii) are widely used in practice. The latter is a less formal criterion, where we take into account the number of weekly downloads on npm [2] and the activity on GitHub (stars, issue management, number of forks and pull requests). Based on these criteria, we chose five static and two dynamic tools for our comparative study (for a summary, see Table 1).

To the best of our knowledge, there is no benchmark designed for comparing JavaScript call graph extraction tools. We, therefore, looked at what inputs other researchers and practitioners use for similar evaluation tasks. One of the benchmarks used by many others was the SunSpider benchmark. The SunSpider benchmark [13] of the WebKit browser engine contains several real-world, single-file JavaScript examples. The benchmark programs are designed to test the WebKit JavaScript engine. As such, these programs contain code of varying complexity, with multiple function types and calls, all contained within single JavaScript files. These characteristics make them an excellent choice for our single-file test subjects.

However, most of the time nowadays, multi-file JavaScript modules are developed that might contain cross-references between the files. To make our study as realistic as possible, we have gathered a set of popular Node.js modules from GitHub, from which we have randomly selected 12 that meet the following criteria: (1) the module consists of multiple JavaScript source files, (2) it is tested well, with at least 75% statement-level coverage (as we also have two dynamic approaches, hence high test coverage is needed), and (3) it is used by at least one hundred other modules. These criteria ensure that our results (based on the selected inputs) are broadly consistent with the results that would be obtained by using the tools in practice.

Our primary intention was to test the tools on inputs (proto-benchmarks) that represent the real-life usage of these tools, hence the results would be useful in practice. Of course, one may want to test these tools with more specific inputs (e.g. domain-specific libraries) or may want to include another tool in the comparison. Keeping that in mind, we have designed

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

this study and our framework to be easily extensible, both with other tools and with other inputs.

A. TECHNICAL OVERVIEW OF THE STUDY PROCESS

Figure 1 displays a high-level overview of the software components employed in our comparative analysis, both external and self-developed. On the test input files (Section III-C), we run each of the selected tools (Section III-B). As can be seen, we had to patch a number of the tools (marked with \) for several reasons (see Section III-B), but the main reason was to extract and dump the built call graphs, which are in the memory of the programs (all the modifications are available in the online appendix package). In the next step, we collected the tools' outputs and ran our data conversion scripts to convert each call graph to a unified, JSON-based format that we specified (see Section III-D for the details). Closure was the only exception, where we had to implement the call graph extraction (as there was no public option for outputting the call graph), so the output is written right into our specified JSON format. In all other cases, we created a custom data parser script that transforms the given tool's arbitrary format into our specified JSON format.

In the case of dynamic call graph creation, the method was a little bit different. We used tools that record all nodes and edges encountered during the execution of a JavaScript program. The first tool, which runs on top of NodeProf (see Section III-B6), creates an event log file containing the newly found nodes and edges and a dynamic call graph is constructed from this event log after the execution is finished. Due to this method, the call graph is always accurate, even if Node.js terminates abnormally. Usually, a program is executed multiple times and a call graph is created for each execution. These call graphs are merged into a final one that contains all nodes and edges found in any of these intermediate graphs. This final call graph is converted to the common JSON format. The second tool we used is a modified Node.js runtime that directly builds the call graph while running the program and dumps it once Node.js terminates.

Using our graph comparison tool, we built a merged JSON with the same structure from the various JSON outputs of the tool results (Section III-E). This combined JSON contains all of the nodes and edges detected by either tool, as well as an additional attribute indicating all of the tool identifiers that found the particular node or edge. On these individual and merged JSON files, we performed our analysis and calculated all of the statistics (all the produced JSON outputs are part of the online appendix package).

B. CALL GRAPH EXTRACTION TOOLS

In this section, we describe the tools we employed in our comparative study.

1) WALA

WALA [37] is a complete framework for both static and dynamic program analysis for Java. It also has a JavaScript front-end based on Mozilla's Rhino parser [12]. In this study, we only employed one of its main components, which is static analysis, call graph creation in particular.

We had to build a driver that serializes the call graph to obtain the results we needed. We utilized an already existing version of the call graph serializer from the official WALA repository for this (*CallGraph2JSON.java*). As a first step, we converted the actual call graph to a basic DOT format, which was then transformed into the final JSON file using our converter script. If the caller function had multiple call sites, WALA generated many edges between the functions. Because our call graph definition allowed only one edge between two functions in one direction, we modified the serializer to filter the edges and merge them if required. We had to address the special case when the call site was in the global scope because there was no explicit caller method in this particular case. As a result, we followed the typical approach of other tools and introduced an artificial `<entry>` node as the source of these edges.

WALA is written entirely in Java, and its primary repository is being actively developed, mostly by the IBM T.J. Watson Research Center. It has been mentioned in over 60 publications [11] since 2003.

2) CLOSURE COMPILER

The Closure Compiler [26] is a real JavaScript compiler. Instead of compiling to machine code, it converts JavaScript to better JavaScript by parsing and analyzing JavaScript applications, removing dead code, rewriting, and compressing the code. It also checks for common JavaScript flaws.

It builds a call graph data structure that is only used internally by other algorithms. As a result, we had to modify the existing source code and add a call graph dumping method (which outputs the internally build call graph into JSON). Closure Compiler contains the artificial root node by default to mimic calls made from the global scope. Closure maintains track of multiple call locations, so the JSON writer filters any duplicate edges to produce an appropriate JSON output for comparison (see Section III-D).

Google is actively developing the Closure Compiler, which is written completely in Java.

3) ACG

ACG (Approximate Call Graph) implements a field-based call graph construction algorithm [36] for JavaScript. The call graph constructor has two basic modes, pessimistic and optimistic, that differ in how interprocedural flows are handled. In our study, we employed both strategies for call graph construction: the default `ONESHOT` (pessimistic) and the `DEMAND` (optimistic). We clarify that ACG has two other strategies, however, the `NONE` strategy does not track interprocedural flow, and the `FULL` strategy is not yet implemented. They yield the same results for simple inputs, however, for bigger Node.js programs, the `DEMAND` strategy tends to find some additional edges when compared to the `ONESHOT` strategy. Whenever we mention ACG without stat-

TABLE 1. Comparison of the used tools (as of 19th January, 2021).

Tool name	Implementation language	Size (SLOC)	Commits	Last commit	Number of contributors	Issues (open/closed)	ECMAScript compatibility
WALA [6]	Java	248,648	6,708	18/01/2021	31	309 (115/194)	ES5
Closure Compiler [4]	Java	481,655	16,249	17/01/2021	448	2666 (820/1846)	ES6 (partial)
ACG [1]	JS	36,317	510	14/06/2019	9	15 (9/6)	ES6 (partial)
npm callgraph [10]	JS	221	31	06/08/2018	2	19 (8/11)	ES6 (partial)
TAJS [14]	Java	236,302	27	23/06/2020	2	15 (6/9)	ES5 (partial)
NodeProf [9]	Java, JS	8,641	426	06/12/2020	8	25 (6/19)	ES2017
nodejs-cg [8]	C, JS	2,172,532	66,637	19/01/2021	376	N/A	ES2020

ing the strategy, that indicates we used it with the default `ONESHOT` setting.

We had to implement the artificial root edge (i.e., `<entry>`) and the filtering of multiple edges for ACG since ACG additionally records and reports edges associated with particular call sites. Furthermore, ACG only reported function line numbers in its output, which we had to enrich with column information. All of these changes are available in the online appendix.

We had to examine all of the forks of the original repository available at the moment and pick the most mature one. We chose the one created by the Persper Foundation.

4) THE NPM CALLGRAPH MODULE

Gunar C. Gessner created npm callgraph, a tiny npm package for creating call graphs from JavaScript code. It parses JavaScript code with UglifyJS2 [24]. Considering its small size and few commits, it is widely used, with over 5,000 downloads. Even though the author does not update the code frequently, he quickly reacts to newly opened issues. During our research, we ran into `TypeError` issues several times. A simple null check solved the problem and we never encountered this kind of error again. We created a fix for this issue and proposed a pull request to the repository, which was already approved and merged to the master branch.[3]

5) TAJS

Type Analyzer for JavaScript [42] is a dataflow analysis tool for JavaScript, developed at Aarhus University, that infers type information and call graphs.

The proposed algorithm is implemented in Java and it is continuously maintained since its initial release. We assume that the linked repository is simply an external mirror of an internal repository that is regularly synced. It was not required to change the source code of TAJS because it had a command line option for dumping the created call graphs into DOT format that we could parse and convert into our unified JSON format.

6) NodeProf

NodeProf [58] is an instrumentation and profiling framework for Node.js modules. This framework is capable of running Node.js modules and providing notifications about certain events in JavaScript code such as function entry and exit, or variable assignment. These notifications can be captured by JavaScript applications called analyses. Our dynamic call graph generator tool[4] is also an analysis, which collects call graph-related information.

Internally, NodeProf uses the Graal JavaScript [5] engine that creates an abstract syntax tree (AST) representation from JavaScript code. NodeProf extends this AST with function calls, which notify analyses about certain events and the Graal JavaScript engine executes the modified AST. These modifications do not change the behavior of a Node.js module so the call graph is accurate. We had to make some modifications in the original source code of NodeProf (mostly for reporting), which can be found on GitHub [9].

7) NODEJS-CG

The other tool we used in this work is called *nodejs-cg* [39], [40], which is a customized Node.js runtime. Node.js uses the V8 [16] engine as the default JavaScript interpreter. It has built-in support for execution tracing. However, using the default tracing mechanism has quite a big overhead, as parsing the output of tracing and building a call graph from it requires a lot of time and space.

Instead, our approach generates the call graph directly, which is faster and requires far less space. The nodes and edges are recorded during the execution of a Node.js application, and the call graph is dumped when Node.js terminates. As the call graph is directly generated by the JavaScript engine without modifying the behavior of a Node.js module, we can conclude that the call graph is accurate in this case too. Our modifications in Node.js can be found on GitHub [8].

8) OTHER STATIC TOOLS WE CONSIDERED

Of course, additional candidate tools might have been included in this study. We noticed several commercial and/or closed-source programs, such as `SAP HANA`, or `JAM` [52]. Nevertheless, we concentrated on open-source tools that are easy to access and can easily be customized to meet our requirements. They are also frequently employed in research and industry.

In this study, we only examined tools that directly support call graph building, either internally or as a public feature. We only examined tools that directly support call graph building, either internally or as a public feature, As a result,

[3]https://github.com/gunar/callgraph/commit/36bab6a0a437c04c2518ae5c4b108791c706eb07

[4]https://github.com/szeged/js-call-graphs

we were forced to exclude some excellent JavaScript analysis tools that do not explicitly support call graph extraction. One such tool was Facebook's open-source solution called `Flow` [34], a prominent static code analysis tool for type checking in JavaScript. Unfortunately, there is no public API for accessing the call graph or the control flow graph that Flow generates. As a result, we would have needed to construct our algorithms on the top of the internal control flow data structure, which would have endangered the study's validity, as the algorithm would have been written by us. The major purpose of this study was to empirically compare existing call graph extraction algorithms, rather than to upgrade all tools to perform call graph extraction.

Additional tools we examined were `JSAI` (JavaScript Abstract Interpreter) [43] and `SAFE` (Scalable Analysis Framework for EcmaScript) [45], both of which can create an intermediate abstract representation from JavaScript on which further analysis can be performed. Certainly, they calculate control and data flow structures, but they employ them specifically for type inference. We were unable to include them in our evaluation study since none of them enable the extraction of call graphs.

The tool `code2flow` [3] appeared to be a good candidate, but because it has been officially abandoned with no follow-up forks, we omitted it from our list. (While the original ACG repository has also been abandoned, there are multiple live forks on GitHub.)

Another reason we omitted potential tools from the comparison was immaturity. We found several projects that had only a few (usually one or two) contributor(s) and had a very brief development period before being abandoned. These tools were hard to use utilize in practice as they lacked documentation. Because of this, we did not take `JavaScript Explorer Callgraph` [7] into consideration. We also took out `callgraphjs` [2] because this project solely provides ACG-related content.

C. COMPARISON SUBJECTS

We established two test input groups for a thorough comparison of the tools.

1) SINGLE FILE BENCHMARK EXAMPLES

As we mentioned earlier, we wanted to include real-world, single-file JavaScript examples that could be analyzed easily either by a program or by hand. We used the SunSpider benchmark [13]. This benchmark consists of 26 JavaScript files with varying complexity that test the WebKit browser engine. A given test file is quite easy to understand, however, it might contain structures that can cause problems for either static or dynamic call graph builders (see for examples, Listings 2 and 7).

2) REAL-WORLD, MULTI-FILE NODE.JS EXAMPLES

We selected several Node.js modules to test the handling of current, ECMAScript 6, and Node.js features (such as module exports or external dependencies, i.e., the *require* keyword) and inter-file dependencies. Unfortunately, the only static tool capable of analyzing such programs was ACG, so we could only employ this tool in the comparison (with both implemented building strategies, `ONESHOT` and `DEMAND`) and the two dynamic approaches. Table 2 summarizes the details of the selected Node.js modules.

TABLE 2. The selected Node.js modules and their size (source lines of code).

Name	Repository URL	SLOC
debug	https://github.com/visionmedia/debug.git	1,083
doctrine	https://github.com/eslint/doctrine	5,109
hessian.js	https://github.com/BugsJS/hessian.js.git	6,796
request	https://github.com/request/request	9,469
express	https://github.com/BugsJS/express.git	11,673
hexo	https://github.com/BugsJS/hexo.git	16,617
karma	https://github.com/BugsJS/karma.git	17,690
bower	https://github.com/BugsJS/bower.git	28,087
shields	https://github.com/BugsJS/shields.git	47,786
pencilblue	https://github.com/BugsJS/pencilblue.git	54,746
jshint	https://github.com/jshint/jshint	68,411
eslint	https://github.com/BugsJS/eslint.git	284,342

D. OUTPUT FORMAT

The tools we chose produce their outputs in specific formats by default. As a result, we needed to process their outputs and convert them into a unified format that is suitable for further analysis. We chose a simple JSON format to store the call graph's nodes and edges.

In the graph, each node has a unique identifier (continuously increasing number), a label (arbitrary string, provided by the tool), and a source code position (created from the name of the file, and line/column information on where the function starts). A function (or node) is identified by its source code position (as identifiers can be different in the results of different tools). Each edge connects exactly two of the nodes by their unique ids.

E. GRAPH COMPARISON

The call graphs were evaluated in two different ways. For the quantitative analysis, we focused on comparing the number of nodes and edges, as well as the similarity of entire call graphs. To assess the quality of the results, we implemented a Python-based call graph comparison script [5] based on the work of Lhoták et al [46]. The script's goal is to detect matching edges identified by various tools. Each node and edge is extended with a new attribute containing a list of tool identifiers that found the particular node or edge. Because many JavaScript functions have no names and relying on a unified unique naming scheme would be cumbersome, nodes and edges are identified using path, line, and column information.

We manually checked the path and line information produced by the evaluated tools to ensure the comparison was

[5]The script is available in our online appendix package: https://doi.org/10.5281/zenodo.7104954

correct. In its standard DOT output, TAJS reported precise line and column information. We implemented or modified the line information extraction in Closure Compiler, WALA, and ACG. Unfortunately, WALA could only report line numbers but no column information, so we had to manually refine the outputs generated by WALA. Because the reported line and column data from npm callgraph was not precise (neither of them), we manually added this information to the produced JSON files. We also implemented a precise line information dump in our dynamic tools.

F. MANUAL EVALUATION

We evaluated all 348 call edges found by the five static and two dynamic tools on the 26 SunSpider benchmark programs as part of the result quality analysis. At least two of the authors performed the manual evaluation by going through all of the edges in the merged JSON files and examining the JavaScript sources to determine the validity of those edges. As a result, we extended the edges of the call graph with a new attribute (called ''valid'') containing whether a particular edge is valid or not (the attribute can be either *true* or *false*). After evaluating the edges, the authors compared their validation results and resolved the two cases where they initially disagreed. The final validated JSON has been created upon consensus.

The large number of nodes and edges in the Node.js modules made manual validation infeasible. To address this, we chose a statistically significant representative random sample of edges (593 in total, see Section IV-B2 for details) with a 95% confidence level and a 5% margin of error. At least two of the authors validated all of these edges in the Node.js sources, and each disagreement was thoroughly discussed. They eventually agreed on whether the selected calls are valid or not.

IV. RESULTS

We divided our findings into two, bigger parts. In the first part, we present the results of the SunSpider benchmark, while in the second part, we report the results of the Node.js modules.

For quantitative analysis, we gathered all the information we could get from the tools, and compared them. Regarding the quality of the results, we used our exact position-based call graph comparison tool that we already introduced in Section III-E. Using our tool, we have identified the call edges found by the different tools. We also compared the amount of common edges found by the arbitrary combination of the tools.

All of the presented diagrams were created with the jvenn [23] diagram creation tool. In the online appendix package, we included the interactive version of all the Venn diagrams we show in this section.

A. SunSpider BENCHMARK RESULTS

As we mentioned beforehand, we used the SunSpider benchmark to assess the basic capabilities of the tools. This benchmark consists of 26 files, representing 26 separate programs. At a time we analyzed one of them. The main idea here was to provide simple programs that the tools can easily analyze. After completing the analyses, we gathered the different outputs and transformed them into our predefined JSON format (see Section III-A).

1) QUANTITATIVE ANALYSIS

Based on the data we obtained, we have produced some basic statistics, which are shown in Table 3.

In the table, we can see the number of nodes (i.e. the number of found functions) and edges (i.e. the number of found calls between two functions) found by the tools for every input. There are several inputs for which each tool reported the same amount of nodes and edges (e.g., math-partial-sums.js, math-spectral-norm.js). For some inputs (e.g., bitops-3bit-bits-in-byte.js, string-validate-input.js), the results are very similar, however, they are not exactly the same. However, it is noteworthy that the used inputs are relatively small (with only few functions and few calls), thus there is only a small room for disagreement. Last but not least, there are also inputs for which the tools reported completely different results (e.g., crypto-md5.js, date-format-tofte.js).

As we can see, we can find a static tool that produces similar results to the dynamic tool in almost every case. For example, all of the tools agreed on the number of nodes and edges for math-spectral-norm.js and string-fasta.js. For bitops-bitwise-and.js and regexp-dna.js, we can see that none of the static or dynamic tools can find a node. Since bitops-bitwise-and.js contains only some statements without calling any function, none of the tools realize a node (or an edge). In the case of regexp-dna.js, we can also see some statements, however, some calls to builtin functions happen. In our research, we do not take into account the builtin function calls.

In order to characterize and express the similarity of call graphs (nodes and edges together) with one measure, we have reviewed some well-known general graph similarity measurement approaches [17], [18], [19], [27], [41]. For our purposes, we chose a relatively simple edit distance measure for graphs [18] as it has an intuitive meaning. This measure is similar to the Levenshtein edit distance [49] defined for strings; the minimal number of insert, delete, or substitute operations required to get two identical graphs.

Table 4 summarizes the average graph edit distances of the call graphs built by the various tools. We measured the above mentioned edit distance for each pair of call graphs on all the 26 programs, then took the average distance values. Therefore, we got a matrix of average distance measures between the tools that is symmetric (as the distance measure itself is symmetric).

The two dynamic tools produced exactly the same results in the case of SunSpider benchmark, so the edit distance between the call graphs is 0. Apart from the two dynamic tools, the largest similarity is between the TAJS static tool and

TABLE 3. SunSpider analysis results.

Benchmark program	Static tools										Dynamic tools			
	npm-cg		**ACG**		**WALA**		**Closure**		**TAJS**		**NodeProf**		**nodejs-cg**	
	nodes	edges	nodes	edges	nodes	edges	nodes	edges	nodes	edges	nodes	edges	nodes	edges
3d-cube	15	23	15	23	16	24	15	23	15	23	15	23	15	23
3d-morph	2	1	2	1	0	0	2	1	2	1	2	1	2	1
3d-raytrace	22	29	28	41	21	22	27	40	28	39	28	39	28	39
access-binary-trees	3	3	4	5	4	5	4	5	4	5	4	5	4	5
access-fannkuch	2	1	2	1	3	2	2	1	2	1	2	1	2	1
access-nbody	8	11	12	15	8	11	11	14	12	15	12	15	12	15
access-nsieve	3	2	3	2	2	1	3	2	3	2	3	2	3	2
bitops-3bit-bits-in-byte	2	1	2	1	3	2	2	1	3	2	3	2	3	2
bitops-bits-in-byte	2	1	2	1	3	2	2	1	3	2	3	2	3	2
bitops-bitwise-and	0	0	0	0	0	0	0	0	0	0	0	0	0	0
bitops-nsieve-bits	3	2	3	2	3	2	3	2	3	2	3	2	3	2
controlflow-recursive	4	6	4	6	4	6	4	6	4	6	4	6	4	6
crypto-aes	17	16	17	16	13	16	17	16	13	14	13	14	13	14
crypto-md5	21	30	21	30	3	2	21	30	12	15	12	15	12	15
crypto-sha1	18	23	18	23	3	2	18	23	9	8	9	8	9	8
date-format-tofte	18	18	19	20	2	1	3	2	3	2	12	11	12	11
date-format-xparb	0	0	14	14	13	17	14	14	5	5	6	5	6	5
math-cordic	5	5	5	5	5	5	5	5	5	5	5	5	5	5
math-partial-sums	2	1	2	1	2	1	2	1	2	1	2	1	2	1
math-spectral-norm	6	6	6	6	6	6	6	6	6	6	6	6	6	6
regexp-dna	0	0	0	0	0	0	0	0	0	0	0	0	0	0
string-base64	3	2	3	2	3	2	3	2	3	2	3	2	3	2
string-fasta	5	4	5	4	5	4	5	4	5	4	5	4	5	4
string-tagcloud	4	4	12	18	2	1	11	17	3	2	6	6	6	6
string-unpack-code	0	0	13	20	5	8	12	64	13	20	13	16	13	16
string-validate-input	4	3	5	4	5	4	5	4	5	4	5	4	5	4
$\sum$	**169**	**192**	**217**	**261**	**134**	**146**	**197**	**284**	**163**	**186**	**176**	**195**	**176**	**195**

either of the two dynamic tools. On average, only 1.58 operations would be needed to convert one call graph to the other, which means these are very similar to each other. It is a bit surprising, as the closest two static tools (ACG and Closure) have an edit distance of 3.58 on average. WALA has the largest distance from all the other tools with the maximum edit distance of 10.42. The dynamic tools have edit distances over 6 for all the tools except TAJS. However, dynamic call graphs are not the ones with the greatest edit distance from all the other graphs.

2) RESULT QUALITY ANALYSIS

The Venn diagram of the call edges found (by only the five static tools) in the 26 benchmark programs is shown in Figure 2. The first numbers show the true positive edges (which is assessed by our manual evaluation, see Section III-F), while the second numbers are the total number of found edges. The percentages below the numbers represent the ratio of true positive edges in that particular area (compared to the total number of edges found by the given combination of the tools). The number (and distribution) of edges found by all possible subsets of the five tools is emphasized in this figure.

348 edges were found by the tools, out of which 184 were true positive edges. All five tools found 98 edges in common, all of which were true positive hits. There is less agreement in the case of remaining edges. Four tools found edges that other tools missed. As for edges found by only one tool (WALA, Closure Compiler and npm callgraph (npm-cg) reported such edges), all of them turned out to be false positive.

a: EDGES FOUND BY NPM-CG ONLY

18 unique edges were found only by the npm-cg tool. All the 18 edges were false positive hits, as we manually validated all of them. The edges (without exception) represent calls from the program's global scope to a given function. Although the callee sites did exist in every case, the caller nodes should have been other functions (and not the global scope). An actual example[6] from the *access-nbody.js* benchmark program is shown in Listing 1.

The reported edge's callee is `Sun()` (line 1 in the listing), while the caller is reported to be the global scope (i.e. <entry>). However, this is not true, an anonymous function (which starts at line 8 in the listing) calls `Sun()`. Yet all of the other tools correctly identified this call.

b: EDGE FOUND BY ACG ONLY

Only one edge, a false positive edge, was discovered by ACG and no other tools. It is a call[7] to a function added to the built-in *Date* object via its prototype property in *date-format-tofte.js*. Even though the call actually exists, the caller func-

[6] <entry>→access-nbody.js:74:13.
[7] date-format-tofte.js:186:15→date-format-tofte.js:8:38.

TABLE 4. The average graph edit distances of the tools on the SunSpider benchmark.

		Static tools					Dynamic tools	
		ACG	Closure	npm-cg	WALA	TAJS	NodeProf	nodejs-cg
Static tools	ACG	0.00	3.58	5.88	9.92	5.27	6.15	6.15
	Closure	3.58	0.00	8.62	10.42	6.23	7.35	7.35
	npm-cg	5.88	8.62	0.00	9.12	8.00	8.65	8.65
	WALA	9.85	10.42	9.12	0.00	5.42	6.54	6.54
	TAJS	5.27	6.23	8.00	5.50	0.00	**1.58**	**1.58**
Dyn.	NodeProf	6.15	7.35	8.65	6.62	**1.58**	0.00	0.00
	nodejs-cg	6.15	7.35	8.65	6.62	**1.58**	0.00	0.00

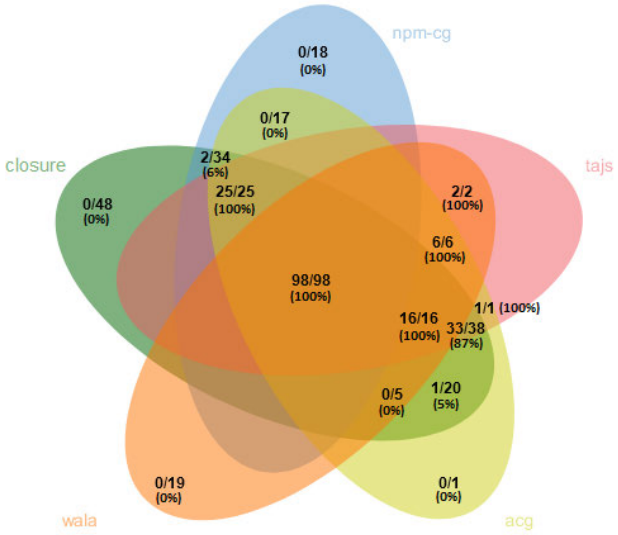

FIGURE 2. Venn diagram of the true/total number of edges found by the tools.

```
function Sun(){
  return new Body(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, SOLAR_MASS);
}
...
var ret = 0;

for ( var n = 3; n <= 24; n *= 2 ) {
  (function(){
    var bodies = new NBodySystem( Array(
      Sun(),Jupiter(),Saturn(),Uranus(),Neptune()
    ));
    ...
  }
}
```

LISTING 1. A false call edge found by npm-cg.

tion cannot be reached from the entry point of the program. Hence the call never happens during the program's execution. Listing 2 shows the excerpt of this call.

c: EDGES FOUND BY WALA ONLY

All 19 of the unique edges found only by WALA are false positive, but for various reasons. WALA was unable to identify the target node of the call edges in 5 cases, resulting in

```
Date.prototype.formatDate = function (input,time) {
  ...
  function W() {
    ...
    var prevNY = new Date("December 31 " + (Y()-1) + "
        00:00:00");
    return prevNY.formatDate("W");
  }
  ...
}

var date = new Date("1/1/2007 1:11:11");

for (i = 0; i < 500; ++i) {
    var shortFormat = date.formatDate("Y-m-d");
    var longFormat = date.formatDate("l, F d, Y g:i:s A");
    date.setTime(date.getTime() + 84266956);
}
```

LISTING 2. A false call edge only found by ACG.

"unknown" as targets. We manually analyzed these cases and discovered that all of the 5 cases are implied by *Array()* calls. Since all built-in and external (library) calls are excluded from the analysis, these edges are clearly false positives.

A group of 10 false positive edges produced by the *date-format-xparb.js* program. This program contains a large switch statement with numerous cases that builds up source code (and calls to various functions) as simple strings. During the run of the program, dynamically created string are being executed using the *eval()* command to extend the prototype of the *Date* object dynamically with generated formatting functions. The dynamically added functions are then called from the *dateFormat* function. WALA recognizes direct edges from *dateFormat* to the functions generated into the body of the formatting functions, which is incorrect, as the functions are called from the dynamically created formatting functions that are called by *dateFormat*.

Invalid recursive call edges indicated in the *string-unpack-code.js* program are the cause of the remaining 4 false positive edges. Although there are several functions with the same name in various scopes, WALA was unable to distinguish between them.

d: EDGES FOUND BY CLOSURE ONLY

The *string-unpack-code.js* program contains each and every one of Closure's unique edges. All these edges are false

```
1  var decompressedMochiKit = function(p,a,c
2      ,k,e,d){e=function(c){return(c<a?"":
3      e(parseInt(c/a)))+((c=c%a)>35?String.
4      fromCharCode(c+29):c.toString(36))}
5      ...
6  }(...);
7  var decompressedDojo = function(p,a,c
8      ,k,e,d){e=function(c){return(c<a?"":
9      e(parseInt(c/a)))+((c=c%a)>35?String.
10     fromCharCode(c+29):c.toString(36))}
11     ...
12 }(...);
```

LISTING 3. A confusing code part from string-unpack-code.js.

```
1  function fast3bitlookup(b) {
2    ...
3  }
4  function TimeFunc(func) {
5    ...
6    for(var y=0; y<256; y++) sum += func(y);
7    ...
8  }
9  sum = TimeFunc(fast3bitlookup);
```

LISTING 4. A true call edge found by WALA and TAJS.

positive ones. The explanation of the false positive edges is quite simple: the program only examines the length of results of various, compressed code snippets; calls are mostly made to built-ins in the codes. Similar to WALA, Closure also seems to ignore the visibility of identifiers within scopes; Listing 3 provides a sketch of the problematic calls.

The inner function redefining parameter *e* of the outer function (line 2 in the listing) is called within itself (line 3 in the listing), which is correctly identified only by Closure and TAJS. However, Closure reports edges from the same location to all the other places where a function *e* is called (e.g. line 9), which is false, because *e* is not the same *e* as it is already in another scope referring to another locally created function denoted by the same name. The majority of the edges detected here would be false anyway because the *string-unpack-code.js* defines four deeply embedded functions with identical parameter names.

e: INTERESTING EDGES FOUND BY TAJS

TAJS did not find any unique edges. Although, it did discover a complex control flow that was only found by ACG (among the static tools). Aside from WALA, TAJS was the only tool that could recognize higher-order function calls. Listing 4 shows such a call[8] in *bitops-3bit-bits-in-byte.js*.

f: PRECISION, RECALL, F-MEASURE

In order to get the most accurate information retrieval metrics as possible, we considered only edges that were reported by the dynamic tools, i.e. that actually happened during the execution of the given program.

We systematically evaluated all 348 call edges the static tools found, and divided them into 3 groups:

- true positive (TP): edge that exists and actually realized during the execution.
- false positive (FP): edge that does not exist in the source code.
- pseudo-positive (PP): edge that could be a real call but stay unrealized due to lack of test inputs of the dynamic analysis.

At least two of the authors evaluated each call edge. Identifying true positive edges was an easy task, however, differentiating between false positive and pseudo-positive edges was a harder task to do. False positive edges do not exist in the source code, but either the edge's caller or callee have a function signature that is similar to a function signature that actually exists in the program. In contrast, in the vast majority of the pseudo-positive cases, the caller function never gets called, thus the call from the caller to callee (which would otherwise be a valid, possible call) is never realized.

During the evaluation of the static analysis, we found 184 true positive edges. We then added all edges that can be found only by the dynamic tools, as they certainly happen during the program's executions. In total, we considered the resulting 195 edges as a golden standard. Then, for each tool and all possible combinations of them, we were able to calculate the well-known information retrieval metrics (precision and recall). We should note that only simple call edges were evaluated and compared; paths along these edges (i.e. call chains) were not taken into account. The impact of missing or extra edges may vary depending on how many paths going through them, and this may affect the precision and recall of the found call chain paths.

The detailed statistics of the tools can be found in Table 5. The first column (Tool) is the name of the tool or combination of tools. The second and third column (TP and FP) shows the total number of true and false positive instances found by the appropriate tool or tool combination. The following column (PP) shows the number of pseudo-positive edges, by which we mean edges that would be real if the program execution reached the caller's side. As this does not happen, these edges are not counted as true positive edges. In the fifth column (All), we display the total number of edges found by the appropriate tool or tool combination. The sixth (Prec.), seventh (Rec.), and eighth (F) columns contain the precision (TP / All), recall (TP / 195[9]) and F-measure values, respectively.

From the individual tools, TAJS stands out with its almost perfect (97%) precision and quite high recall (93%) values. While ACG and Closure have quite high recall (very close to that of TAJS), their precisions are far below TAJS's. Closure achieved the worst precision (62%), while WALA had the lowest recall (63%) among the tools. ACG found the highest number of true positive edges, however, it found several pseudo-positive edges, thus explaining the rather low precision and recall.

[8] bitops-3bit-bits-in-byte.js:28:18→bitops-3bit-bits-in-byte.js:7:24.

[9] The total number of true positive edges.

TABLE 5. Precision and recall measures for individual tools and their combinations.

Tool	TP	FP	PP	All	Prec.	Rec.	F
ACG	182	6	73	261	70%	93%	80%
Closure	175	54	55	284	62%	90%	73%
npm-cg	125	18	49	192	65%	64%	65%
TAJS	181	4	1	186	97%	93%	95%
WALA	122	19	5	146	84%	63%	72%
ACG+Closure	182	54	73	309	59%	93%	72%
ACG+npm-cg	182	24	73	279	65%	93%	77%
ACG+TAJS	184	6	73	263	70%	94%	80%
ACG+WALA	184	25	73	282	65%	94%	77%
Closure+npm-cg	175	72	72	319	55%	90%	68%
Closure+TAJS	184	54	55	293	63%	94%	75%
Closure+WALA	183	73	55	311	59%	94%	72%
npm-cg+TAJS	183	22	50	255	72%	94%	81%
npm-cg+WALA	149	37	54	240	62%	76%	69%
TAJS+WALA	181	23	6	210	86%	93%	89%
ACG+Closure+npm-cg	182	72	73	327	56%	93%	70%
ACG+Closure+TAJS	184	54	73	311	59%	94%	73%
ACG+Closure+WALA	184	73	73	330	56%	94%	70%
ACG+npm-cg+TAJS	184	24	73	281	65%	94%	77%
ACG+npm-cg+WALA	184	43	73	300	61%	94%	74%
ACG+TAJS+WALA	184	25	73	282	65%	94%	77%
Closure+npm-cg+TAJS	184	72	72	328	56%	94%	70%
Closure+npm-cg+WALA	183	91	72	346	53%	94%	68%
Closure+TAJS+WALA	184	73	55	312	59%	94%	73%
npm-cg+TAJS+WALA	183	41	55	279	66%	94%	77%
ACG+Closure+npm-cg+TAJS	184	72	73	329	56%	94%	70%
ACG+Closure+npm-cg+WALA	184	91	73	348	53%	94%	68%
ACG+Closure+TAJS+WALA	184	73	73	330	56%	94%	70%
ACG+npm-cg+TAJS+WALA	184	43	73	300	61%	94%	74%
Closure+npm-cg+TAJS+WALA	184	91	72	347	53%	94%	68%
ALL	184	91	73	348	53%	94%	68%

Looking at the two tools combinations, TAJS+WALA stand out based on F-measure (89%). However, this combination did found only true positive edges that TAJS found, WALA did not add any unique edge, so using TAJS only would yield a better result than using both of the tools. The second best F-measure value was produced by npm-cg+TAJS. It has a worse F-measure than TAJS+WALA, but interestingly this combination find more edges than TAJS+WALA. The next best F-measure value (80%) was produced by using ACG and TAJS together. Interestingly, they found the most true positive edges (184 out of 195) any tool or tool combination could find, while maintaining quite high F-measure. This combination found the highest number of pseudo-positive edges too. In fact, using TAJS seems to be the best choice, followed by the combination of TAJS and WALA, as their F-measures are the highest. Taking pseudo-positive edges into account (as if they would true positives), either ACG or TAJS could be a good choice, and combining them would seem to be the best we could establish in this context. Taking all the tools into consideration, the combined precision decreases to 53% with a recall value of 94%.

Answer to RQ1: Our manual validation shows that the static call graph extractor tools do not find exactly the same edges. This is due to the details in implementation and the underlying concept of the tools (differences in handling `eval()`, visibility of identifiers, or identifying call sites). These differences all contribute to the different results that the tools produce.

Comparison of static and dynamic results.

Table 6 summarizes the relation between the call edges found by the static call graph tools and the dynamic call graph extraction process. We kept only those static edges that were evaluated to be true positive instances. As can be seen, there is quite a big variance in the intersections and differences in the call edges among the static tools. The two extremes are npm-cg and TAJS. On the one hand, npm-cg misses 70 valid edges and has the lowest intersection (125 edges) with the dynamic approach. TAJS, on the other hand, produces a result that is very similar to that of the dynamic approach. 99% of the edges found by TAJS is also in the dynamic call set and TAJS also finds 93% of all dynamic edges (i.e., misses only 14 edges found by NodeProf).

Interestingly, WALA produced only 5 edges that are not in the dynamic set (i.e., 96% precision based on the dynamic edges), while Closure found 90% of the dynamic edges at the price of introducing 55 edges not in the dynamic set. We manually checked all the edges that were found only by a static tool or only by the dynamic approaches.

g: EDGES FOUND ONLY BY THE DYNAMIC APPROACHES

Given the highly dynamic nature of JavaScript, it is no surprise that there were several edges found by only the dynamic tools. All of these edges were valid calls between functions, but they are mostly undetectable by a static analyzer.

For example, in Listing 5, an anonymous function dynamically adds several functions to its parameter

TABLE 6. Comparison of static and dynamic edges.

Tool	Static only	Static ∩ Dynamic	Dynamic only	Precision_{dyn}	Recall_{dyn}
ACG	71	162	33	0.70	0.83
Closure	55	175	20	0.76	0.90
npm-cg	49	125	70	0.72	0.64
TAJS	1	181	14	0.99	0.93
WALA	5	122	73	0.96	0.63

```
1   (function (s) {
2     // ...
3     s.parseJSON = function (filter) {
4       // ...
5       function walk(k, v) { // line 180
6         // ...
7         return filter(k, v);
8       }
9       // ...
10      return typeof filter === 'function' ? walk('', j) : j;
11      }
12      // ...
13    };
14  // ...
15  })(String.prototype);
16 // ...
17 var tagInfoJSON = 'A long string on line 226 in string—
       tagcloud.js';
18 // ...
19 var tagInfo = tagInfoJSON.parseJSON(function(a, b) { /*code*/
       }); // line 229
```

LISTING 5. A call detected only by dynamic tools.

called `s`. The function is immediately called with the `String.prototype` parameter meaning that every String object will be extended with the defined functions and properties. Hence, `tagInfoJSON` (which is a string) will have a `parseJSON` function that takes a function as an argument. The function call[10] was realized by an inner function called inside `parseJSON`, `walk`, which calls the `parseJSON`'s parameter named `filter`. Since `parseJSON` was added dynamically, it would have been hard to detect by static analysis alone.

Dynamic evaluation of strings is also a typical case which is hard to detect by any static analyzer while a dynamic tool can find it easily. In the case[11] depicted in Listing 6, the program adds a `formatDate` function to all Date instances in the program. The function `formatDate` splits the desired output format (a parameter called `input`), and iterates through it. If it finds a format character presented in the predefined variable `switches`, the function calls the corresponding function with `eval`. While it is a pretty straightforward dynamic call, it is really hard to detect with a static analysis tool.

As we mentioned before, in *string-unpack-code.js*, the dynamic tools found more nodes than any of the static tools. We evaluated these nodes too, which are proved to be valid and existing functions. The static analyzers missed these nodes because they are not calling any functions (as they are callback functions that return with an element of an array).

[10] string-tagcloud.js:180:26→string-tagcloud.js:229:45.
[11] date-format-tofte.js:8:38→date-format-tofte.js:83:15.

h: EDGES FOUND ONLY BY STATIC TOOLS

In accordance with our expectations, there were some edges found only by the static tools. Usually, the static call graphs contain possible call edges that are never realized during run time. We must note however, that in the case of the SunSpider benchmark, there is a fair amount of dead code, which causes lots of unrealized but possible calls. This sheds light to one of the weaknesses of the dynamic approach, namely that an insufficient test input makes the call graph imprecise. Nonetheless, there are several possible edges that are unrealized due to some condition on the inputs and we ran the dynamic analysis with only one input vector provided with the tests.

For example, in Listing 7, there is a function call[12] to `String.escape` (line 19). This call is never realized in practice as the function `dateFormat` was never called with a parameter containing a backslash.

Answer to RQ2: Static call graph extractor tools found a significant proportion of the real edges detected by dynamic analysis on the SunSpider benchmark. However, most static approaches introduce false positive edges. In addition, they often miss real edges. According to our manual validation, the majority of the missed edges come from dynamic calls, which in most cases would be extremely difficult to detect from the source code alone (i.e., using static analysis).

B. NODE.JS MODULE RESULTS

1) QUANTITATIVE ANALYSIS

To evaluate the practical capabilities of the selected tools, we analyzed 12 real-world, popular open-source Node.js modules. We had more candidates, but these were the ones we could analyze both dynamically (i.e. that had the proper amount of executable tests) and statically without errors. Details about the subject programs can be found in Section III-C2.

Unfortunately, npm callgraph and WALA were unable to analyze whole, multi-file projects because they cannot resolve calls among different files (e.g., requiring a module).

In earlier stages of our work [21], we were able to use Closure Compiler too, which seemed to be quite imprecise in terms of real, Node.js applications. A manual evaluation on a sample of 240 edges found by Closure on various Node.js modules showed only 40 real call edges, which is less than 20% precision. Moreover, the developers of Closure

[12] date-format-xparb.js:26:32→date-format-xparb.js:347:25.

```
function arrayExists(array, x) {
  for (var i = 0; i < array.length; i++) {
    if (array[i] == x) return true;
  }
  return false;
}
Date.prototype.formatDate = function (input,time) {
  var switches = ["...", "g", "G", "..."];
  // ...
  function g() {/* 12 hour format of the given date */}
  // ...
  var ia = input.split("");
  var ij = 0;
  while (ia[ij]) {  // this will be "g"
    //...
    if (arrayExists(switches,ia[ij])) {  // "g" in switches
      ia[ij] = eval(ia[ij] + "()");  // ia[ij] = g()
    }
    ij++;
  }
  // ...
}
var date = new Date("1/1/2007 1:11:11");
var longFormat = date.formatDate("g");
```

LISTING 6. A call detected only by dynamic tools.

```
Date.formatFunctions = {count:0};

Date.prototype.dateFormat = function(format) {
  if (Date.formatFunctions[format] == null) {
    Date.createNewFormat(format);
  }
  // ...
}

Date.createNewFormat = function(format) {
  // ...
  for (var i = 0; i < format.length; ++i) {
    ch = format.charAt(i);
    if (!special && ch == "\\") {
      special = true;
    }
    else if (special) {
      special = false;
      code += "'" + String.escape(ch) + "' + ";  // this call
            is never realized, but is a valid possible call
    }
    // ...
}
// ...
String.escape = function(string) {
  return string.replace(/('|\\)/g, "\\%$1");
}
// ...
var date = new Date("1/1/2007 1:11:11");
for (i = 0; i < 4000; ++i) {
  var shortFormat = date.dateFormat("Y-m-d");
  var longFormat = date.dateFormat("l, F d, Y g:i:s A");
  date.setTime(date.getTime() + 84266956);
}
```

LISTING 7. An unrealized edge.

Compiler have removed the explicit call graph data structure support from their tool.[13] Hence, in the current stage of our research, we decided to omit Closure Compiler from the comparison as well. TAJS supports the require command, nonetheless it was still unable to detect call edges in multi-file Node.js projects. Therefore, we could apply only ACG as a static tool to recognize call edges in Node.js modules. Thus, we used only this static and the two dynamic tools to perform the analysis and comparison on the selected Node.js modules.

To gather as much information as we can, we performed the analysis using both applicable strategies that ACG offers (ONESHOT and DEMAND). The ONESHOT strategy tracks the inter-procedural flow but it tracks only for one-shot closures that are invoked immediately. The DEMAND strategy (called optimistic approach) performs inter-procedural propagation along the edges that may end at a call site.

As for the NodeProf dynamic analysis, we had to execute all the tests provided as part of the Node.js modules and run our NodeProf-based analysis tool. The test systems of the selected programs use the *npm* tool of Node.js, which spawns Node.js binaries with various command line arguments. The biggest challenge of the implementation was correctly supporting all of these arguments.

We calculated some basic statistics from the gathered data that is shown in Table 7. The table displays the number of nodes (functions) and edges (possible calls between two functions) found by the tools. As can be seen, the results show resemblance, the correlation between nodes and edges found by the tools (and approaches) is high. Unsurprisingly, there are no exact matches in the number of nodes and edges for such complex input programs. The two dynamic tools produced almost identical results in terms of the numbers of nodes and edges. Although in one case, there is a slight difference between the found edges (eslint).

2) RESULT QUALITY ANALYSIS

During the evaluation of the edges, we could only validate the existance of the edges, due to the huge size of the input programs. So it is possible that we labeled pseudo-positive edges as true positive edges, since checking whether the execution of such huge programs reaches a particular point (in any way) is cumbersome.

As we already mentioned earlier, we could use only ACG (with two strategies), NodeProf and Nodejs-cg to analyze the state-of-the-art Node.js modules listed in Table 2. Figure 3 shows a Venn diagram of the results. Taking every tool into consideration, only 6,818 edges were found by all of the tools, which is approximately 8% of all edges.

While ACG ONESHOT did not report any edge that was missed by others, ACG DEMAND reported 43,009 unique edges that were found by only this strategy. Interestingly, it also found 169 edges that were detected by the dynamic approaches (and missed by the other static strategy). The DEMAND strategy improves the recall of ACG, at the price of significantly lowering precision (see Table 8). ACG ONESHOT and ACG DEMAND reported 29,227 edges that were not reported by any of the dynamic tools, but both static strategies found them. This can be easily explained by the fact that the ACG DEMAND strategy is not an entirely new algorithm, but rather extends ACG ONESHOT's results. This is the explanation for ACG ONESHOT not finding any unique

[13] https://github.com/google/closure-compiler/commit/5d6c9326f3d8a2255839be439cfb2713d06a60f7.

TABLE 7. Node.js analysis results.

	Static tools				Dynamic tools			
	ACG ONESHOT		ACG DEMAND		NodeProf		Nodejs-cg	
Node module	nodes	edges	nodes	edges	nodes	edges	nodes	edges
bower	674	2146	710	2464	790	1177	790	1177
debug	32	29	35	33	22	23	22	23
doctrine	87	179	87	179	92	195	92	195
eslint	2529	13139	2545	13436	3646	6658	3646	6660
express	122	262	133	506	176	345	176	345
hessian	81	201	81	201	117	224	117	224
hexo	440	1173	482	2922	927	1351	927	1351
jshint	351	1019	378	1128	262	360	262	360
karma	443	781	449	817	510	751	510	751
pencilblue	2192	8862	2443	48512	863	1126	863	1126
request	114	217	114	218	169	233	169	233
shields	1410	8062	1524	8832	1609	2236	1609	2237
$\sum$	**8475**	**36070**	**8981**	**79248**	**9183**	**14679**	**9183**	**14682**

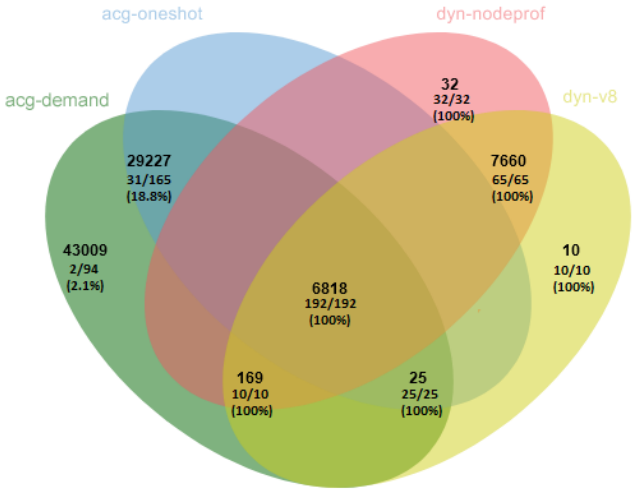

FIGURE 3. Venn diagram of the edges found by the tools on the 12 Node.js modules combined.

edges. Since the amount of edges here is orders of magnitude larger than in the case of the SunSpider benchmarks, we were not able to entirely validate all the calls manually.

However, we evaluated a statistically significant amount of random samples. To achieve a 95% confidence level with a 5% margin of error, we evaluated 593 edges in total. Particularly, we evaluated 259 edges found by ACG only (94 found by the DEMAND strategy only and 165 found by both ONESHOT and DEMAND), 227 edges found by ACG and either of the dynamic approaches (192 from the intersection of all four approaches, 10 from the intersection of ACG DEMAND and the dynamic tools, and 25 from the intersection of ACG ONESHOT and DEMAND and Nodejs-cg dynamic tool). We also evaluated 65 edges found by both dynamic but neither of the static tools. We evaluated all edges found by only one of the dynamic tools.

From 259 examined edges found by only the two static strategies, 33 were valid (2 were found by ACG DEMAND, 31 were found by both strategies). The rest (226 examined edges) were not valid edges, of which 92 were found by ACG DEMAND, and 134 were found by both strategies. As we can see, approximately one fifth (18.8%) of the edges found by the ACG ONESHOT strategy (which is a subset of ACG DEMAND) but not the dynamic approach is true. If we look at all the edges found by ACG DEMAND, the ratio falls under 13%, so using the ACG DEMAND strategy increases the number of true positive edges found, however it increases the number of false positive edges even more. Therefore the true positive rate is slightly above 12.7% for our sample, which means that we are 95% sure that around one eighth of the edges found by only ACG but not the dynamic approach are true edges.

We evaluated 192 edges that were found by all of the strategies, all of which were valid, true positive edges. On the one hand, 382 edges were found by both static approaches, of which 248 were true positive edges, resulting in a true positive rate of 64.92%. On the other hand, we evaluated 267 call edges that were found by both dynamic approaches, all of them were valid. We also evaluated the edges found by one of the dynamic tools, all of which were true positives. Not surprisingly, all dynamic edges were valid, but we wanted to double-check to make sure we did not introduce any errors in our dynamic analysis tools.

Based on the evaluated samples, we can even give an estimation to the precision and recall of each approach. Table 8 shows these estimated numbers. We estimate the precision of approaches with the TP rate of the evaluated samples. For the estimated recall values, we need the expected number of total valid edges, which we approximated by the number of edges found by the dynamic approach (as all the evaluated samples are true for such edges) plus the TP rate proportion of edges found only by the static approaches (18.18% of 29,227 edges plus 2,21% of 43,009 edges), which yields 6,406. As it can be seen, we rounded the values to the greatest integer that is less or equal to the value we got. Based on the TP rates for individual approaches, we could estimate the

TABLE 8. Precision and recall values.

Tool(s)	Prec.	Rec.	F
ACG ONESHOT	34.20%	58.40%	43.13%
ACG DEMAND	16.93%	63.53%	26.74%
Dynamic (NodeProf)	100.00%	69.50%	82.01%
Dynamic (Nodejs-cg)	100.00%	69.52%	82.02%
ACG ONESHOT+Nodejs-cg	45.94%	95.52%	62.04%
ACG ONESHOT+NodeProf	45.94 %	95.50%	62.04%
ACG DEMAND+Nodejs-cg	29.91%	99.85%	46.04%
ACG DEMAND+NodeProf	29.91%	99.83%	46.03%
ACG ONESHOT+Nodejs-cg+NodeProf	59.49%	95.67%	73.36%
ACG DEMAND+Nodejs-cg+NodeProf	39.39%	100%	56.52%
ACG ONESHOT+DEMAND	16.93%	63.53%	26.74%
Nodejs-cg+NodeProf	100%	69.67%	82.12%
ALL	38.10%	100%	55.17%

expected number of true edges found by approaches: 12,334 for ACG ONESHOT (1*6818 + 1*25 + 0.1818*29227), 13,418 (1*6818 + 1*169 + 1*25 + 0.0212*43009) for ACG DEMAND, 14,679 (1*6818 + 1*32 + 1*7660 + 1*169) for NodeProf, and 14,682 (1*6818 + 1*7660 + 1*10 + 1*25 + 1*169) for the Nodejs-cg tool. The ratios of these numbers will give us a rough estimate of the tools' recalls. Obviously, the dynamic approaches have perfect precision, however, they miss several edges probably due to the insufficient number of test inputs, thus having recall values of 69.50% for NodeProf and 69.52% for Nodejs-cg. ACG ONESHOT has a greater F-measure value than the DEMAND strategy. Despite the DEMAND strategy has a bit higher recall value but its precision is about the half that of the ONESHOT strategy.

While we were evaluating the results, we noticed several edges (found only by the dynamic tools) that in our opinion could have been detected by a static algorithm. A typical example for a weakness of ACG is when a function is being called within an export statement, like in Listing 8 (taken from express.js[14]).

But this is not the only case when the static approach missed a possible edge. According to our experiences, there are circumstances when both strategies miss edges that it otherwise recognizes correctly in the majority of cases. An example of a missed edge is shown in Listing 9 (taken from JSHint[15]). This suggests that there might be missed edges due to implementation/technical issues and not just due to the conceptual barriers of the underlying extraction algorithms.

Answer to RQ3: Based on the manually validated samples from real-world Node.js projects, we found that static approaches find roughly 10-20%[16] true positive edges that were missed by dynamic approaches. This is probably due to the incomplete test coverage of the study projects and that they might contain unreachable source code. There may be also tests that we were unable to execute because they required specific hardware or software requirements.

[14] lib/utils.js:1:1→lib/utils.js:274:30.

[15] src/name-stack.js:43:37→src/name-stack.js:8:16.

[16] Compared to the total number of edges found by static approaches only.

```
1 exports.etag = createETagGenerator({ weak: false })
2 // ...
3 function createETagGenerator (options) {
4   // ...
5 }
```

LISTING 8. A typically missed edge that could be detected statically.

C. DISCUSSION OF THE RESULTS

Each approach and tool has advantages and disadvantages. During this comparative study, we distilled the following "lessons".

- Static tools handle recursive calls well; Closure Compiler appears to be the most mature in this respect.
- Edges pointing to nested functions (function in a function) are not handled well by every static tool, e.g., WALA produces a lot of false edges because of this.
- Apart from the dynamic tools, only WALA, TAJS, and ACG with the optimistic strategy (DEMAND) can detect calls of function arguments (i.e. higher-order functions).
- ACG and TAJS cal follow more complex control flows and detect non-trivial call edges.
- Closure often relies only on name matching, which can cause false or missing edges.
- WALA can analyze *eval()* constructs and dynamically built calls from strings to some extent.
- npm-cg mistreats calls from anonymous functions defined in the global scope, meaning it reports the call coming directly from the global scope (instead of the anonymous function).
- TAJS produced the most similar call graph to that of the dynamic analysis.
- From the available static approaches, only ACG is practically suitable for analyzing up-to-date Node.js modules (due to language support and precision).
- Both static and dynamic approaches found true edges that the other missed.

As can be seen, dynamic approaches have perfect precision, since they only report call edges that actually occur. This is also their biggest disadvantage, as they require very high test coverage to have the highest possible recall value. Moreover, there may be code where its execution depends on

```
1  Object.defineProperty(NameStack.prototype, "length", {
2    get: function() {
3      return this._stack.length;
4    }
5  });
6  // ...
7  NameStack.prototype.infer = function() {
8    var nameToken = this._stack[this.length - 1];
9    // ...
10 }
```

LISTING 9. A missed edge that is sometimes recognized statically.

the current operating system, some environment variables, or even the availability of another service, for which traditional unit tests may not be sufficient, and more complex test cases may require multiple environments with different settings (or even different interpreter), which can greatly affect the performance of producing an accurate call graph.

V. THREATS TO VALIDITY

Several factors may have had an impact on our study. We tried to mitigate every possible threat as much as we could.

A. TOOLS

First and foremost, there's a chance that some inconsistencies were caused by our changes to the call graph extraction tools. However, the most of the modifications we made affected the reporting of edges; as a result, their impact is minimal, if any. Another possible threat could have been our dynamic tools which we have modified. The developers of the tools are experts in this field, and have several years of experience. Despite that, we did a thorough code review on these tools before using them. Furthermore, a similar study in the field of dynamic analysis was executed, and the results were previously published [39], [40]. All things considered, we believe that this possible threat does not affect our study.

B. MISSED CANDIDATES

We may have missed some good candidate tools from the comparison, but we made every effort to find any tool that met our requirements. Regardless of this, we think the evaluation strategy and the results of this study are helpful. Furthermore, a comparative study like this may always be replicated and extended.

C. SUBJECTIVITY OF MANUAL EVALUATION

The subjectivity of the evaluators might potentially represent a threat to the manual evaluation of the call edges. By having at least two authors validate each of the selected edges (all edges in the 26 SunSpider benchmark and randomly selected 593 edges in the case of the 12 Node.js programs), we attempted to mitigate this. There were just a few cases when there were early disagreements between the evaluators. They could ultimately come to an agreement. Therefore, we believe that the bias brought on by evaluation errors is negligible.

During the evaluation of the Node.js results, we could only validate the existance of the edges, due to the huge size of the input programs. So it is possible that we labeled pseudo-positive edges as true positive edges. Of course, edges found by a dynamic approach were certainly executed during the run of a program. However, validating edges that were found by only a static approach is a more difficult task, since all execution paths of the program that lead in any way to a given point would have to be examined.

VI. CONCLUSION

JavaScript code analysis has become increasingly popular in the past years. The call graph is a vital structure and the basis for many algorithms for vulnerability analysis, coding issue detection, and type inference.

This paper presents the results of a comparison study of five state-of-the-art static algorithms and two dynamic ones for constructing JavaScript call graphs on 26 WebKit SunSpider benchmark programs and 12 real-world Node.js modules. Our goal was not to declare a winner, but rather to get empirical insights into the capabilities and efficacy of state-of-the-art static call graph extractors, as well as how they compare to dynamic approaches.

Each tool and analysis approach had advantages and disadvantages. For example, Closure Compiler detected calls (mainly recursive calls) that were missed by other static tools, however, it found several false positive edges due to shallow name matching. ACG followed more complicated control flows to detect call edges, resulting in a higher recall value, and maintaining an acceptable precision, but it missed higher-order function calls (callbacks).

It is noteworthy that ACG was the only tool capable of analyzing real-world Node.js modules. WALA could detect higher-order function calls, but it created numerous false positive edges with unknown nodes and had the lowest recall among the tools. The npm callgraph module had quite a low F-measure (as both precision and recall were low) and it did not find any true positive edges others missed. However, its implementation is quite simple and very easy to understand if someone wishes to modify it. TAJS provided great results, it had the highest precision and recall values. Not surprisingly, it produced the most similar call graph to that of the dynamic approach.

The dynamic call graphs were precise, however, they missed certain static edges due to the lack of sufficient test inputs. The results also show that the combined power of numerous tools outperforms that of individual call graph extractors. As a result, we believe that combining static and dynamic approaches in some clever way might result in significant improvements in the quality of the created call graphs. Thus, we encourage both developers and researchers to combine state-of-the-art static and dynamic methods to achieve the best possible results.

REFERENCES

[1] *ACG*. Accessed: Sep. 21, 2022. [Online]. Available: https://github.com/Persper/js-callgraph

[2] *Build Approximate Call Graphs for JavaScript*. Accessed: Jul. 21, 2022. [Online]. Available: https://github.com/asgerf/callgraphjs.dart

[3] *Code2Flow: Pretty Good Call Graphs for Dynamic Languages*. Accessed: Jul. 21, 2022. [Online]. Available: https://github.com/scottrogowski/code2flow

[4] *Google Colsure Compiler*. Accessed: Sep. 21, 2022. [Online]. Available: https://github.com/google/closure-compiler

[5] *GraalVM JavaScript: A High Performance Implementation of the JavaScript Programming Language*. Accessed: Jul. 21, 2022. [Online]. Available: https://github.com/oracle/graaljs

[6] *IBM WALA*. Accessed: Sep. 21, 2022. [Online]. Available: https://github.com/wala/WALA

[7] *The JavaScript Explorer Callgraph Tool*. Accessed: Jul. 21, 2022. [Online]. Available: https://github.com/shrivastava-apurva/Javascript-Explorer—Callgraph

[8] *NodeJS Callgraph*. Accessed: Sep. 21, 2022. [Online]. Available: https://github.com/szeged/node/tree/9a29b05329fbd5521013b97839b0ff0d6ccc6d3e

[9] *NodeProf*. Accessed: Sep. 21, 2022. [Online]. Available: https://github.com/szeged/nodeprof.js/tree/7edcd7a9c17dd061aa307ddbd74fe889af0e7096

[10] *NPM Callgraph*. Accessed: Sep. 21, 2022. [Online]. Available: https://github.com/gunar/callgraph

[11] *Partial List of Publications That Rely on the WALA Infrastructure—WalaWiki*. Accessed: Jul. 21, 2022. [Online]. Available: http://wala.sourceforge.net/wiki/index.php/Publications

[12] *Rhino is an Open-Source Implementation of JavaScript Written Entirely in Java*. Accessed: Jul. 21, 2022. [Online]. Available: https://github.com/mozilla/rhino

[13] *SunSpider 1.0.2 Benchmark*. Accessed: Jul. 21, 2022. [Online]. Available: https://github.com/WebKit/webkit/tree/main/PerformanceTests/SunSpider/tests/sunspider-1.0.2

[14] *TAJS (Type Analyzer for JavaScript)*. Accessed: Sep. 21, 2022. [Online]. Available: https://github.com/cs-au-dk/TAJS

[15] *The State of the Octoverse*. Accessed: Jul. 21, 2022. [Online]. Available: https://octoverse.github.com/

[16] *V8 JavaScript Engine*. Accessed: Jul. 21, 2022. [Online]. Available: https://v8.dev/

[17] L. A. Zager and G. C. Verghese, "Graph similarity scoring and matching," *Appl. Math. Lett.*, vol. 21, no. 1, pp. 86–94, Jan. 2008.

[18] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, and P. Martineau, "An exact graph edit distance algorithm for solving pattern recognition problems," in *Proc. Int. Conf. Pattern Recognit. Appl. Methods*, Lisbon, Portugal, 2015, pp. 271–278.

[19] L. Akoglu, H. Tong, and D. Koutra, "Graph based anomaly detection and description: A survey," *Data Mining Knowl. Discovery*, vol. 29, no. 3, pp. 626–688, May 2015.

[20] K. Ali and O. Lhoták, "Application-only call graph construction," in *ECOOP 2012—Object-Oriented Programming*, J. Noble, Ed. Berlin, Germany: Springer, 2012.

[21] G. Antal, P. Hegedus, Z. Toth, R. Ferenc, and T. Gyimothy, "Static JavaScript call graphs: A comparative study," in *Proc. IEEE 18th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2018, pp. 177–187.

[22] G. Antal, Z. Tóth, P. Hegedus, and R. Ferenc, "Enhanced bug prediction in JavaScript programs with hybrid call-graph based invocation metrics," *Technologies*, vol. 9, no. 1, 2021, doi: 10.3390/technologies9010003.

[23] P. Bardou, J. Mariette, F. Escudié, C. Djemiel, and C. Klopp, "Jvenn: An interactive venn diagram viewer," *BMC Bioinf.*, vol. 15, no. 1, p. 293, Dec. 2014.

[24] M. Bazon. *UglifyJS: JavaScript Parser/Mangler/Compressor/Beautifier Toolkit*. Accessed: Jul. 21, 2022. [Online]. Available: https://github.com/mishoo/UglifyJS/releases/tag/v2.8.29

[25] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 419–429.

[26] M. Bolin, *Closure: The Definitive Guide: Google Tools to Add Power to Your JavaScript*. Sebastopol, CA, USA: O'Reilly Media, 2010.

[27] X. Chen, H. Wang, S. Qiang, Y. Wang, and Y. Jin, "Discovering and modeling meta-structures in human behavior from city-scale cellular data," *Pervas. Mobile Comput.*, vol. 40, pp. 464–479, Sep. 2017.

[28] J. Dijkstra, "Evaluation of static JavaScript call graph algorithms," Ph.D. thesis, Software Analysis and Transformation, Centrum Wiskunde Inform., Amsterdam, The Netherlands, 2014.

[29] M. Dmitriev, "Profiling Java applications using code hotswapping and dynamic call graph revelation," *ACM SIGSOFT Softw. Eng. Notes*, vol. 29, no. 1, pp. 139–150, Jan. 2004.

[30] F. Eichinger, K. Böhm, and M. Huber, "Mining edge-weighted call graphs to localise software bugs," in *Machine Learning and Knowledge Discovery in Databases*. Berlin, Germany: Springer, 2008, pp. 333–348.

[31] F. Eichinger, V. Pankratius, P. W. Große, and K. Böhm, "Localizing defects in multithreaded programs by mining dynamic call graphs," in *Testing—Practice and Research Techniques*. Cham, Switzerland: Springer, 2010, pp. 56–71.

[32] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding program comprehension by static and dynamic feature analysis," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Nov. 2001, pp. 602–611.

[33] F. E. Allen, "Interprocedural data flow analysis," in *Information Processing*. Amsterdam, The Netherlands: North-Holland Publishing Co., 1974, pp. 398–402.

[34] *FaceBook: Flow: A Static Type Checker for JavaScript*. Accessed: Jul. 21, 2022. [Online]. Available: https://flow.org/

[35] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip, "Tool-supported refactoring for Javascript," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 119–138, Oct. 2011.

[36] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for JavaScript IDE services," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA, May 2013, pp. 752–761.

[37] S. Fink and J. Dolby. *WALA—The TJ Watson Libraries for Analysis*. Accessed: Jul. 21, 2022. [Online]. Available: http://wala.sourceforge.net/wiki/index.php/Main_Page

[38] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, 1982.

[39] Z. Herczeg and G. Lóki, "Evaluation and comparison of dynamic call graph generators for JavaScript," in *Proc. 14th Int. Conf. Eval. Novel Approaches Softw. Eng.*, 2019, pp. 472–479, doi: 10.5220/0007752904720479.

[40] Z. Herczeg, G. Lóki, and A. Kiss, "Towards the efficient use of dynamic call graph generators of node.js applications," in *Proc. Int. Conf. Eval. Novel Approaches Softw. Eng.* Cham, Switzerland: Springer, 2019, pp. 286–302.

[41] G. Jeh and J. Widom, "SimRank: A measure of structural-context similarity," in *Proc. 8th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2002, pp. 538–543.

[42] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *Proc. 16th Int. Static Anal. Symp. (SAS)*, vol. 5673. Cham, Switzerland: Springer, 2009, pp. 238–255.

[43] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "JSAI: A static analysis platform for Javascript," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, New York, NY, USA, Nov. 2014, pp. 121–132.

[44] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *J. Comput. Virol.*, vol. 7, no. 4, pp. 233–245, Nov. 2011.

[45] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript," in *Proc. 19th Int. Workshop Found. Object-Oriented Lang.*, 2012, p. 96.

[46] O. Lhoták, "Comparing call graphs," in *Proc. 7th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng.*, Jun. 2007, pp. 37–42.

[47] M. Madsen, B. Livshits, and M. Fanning, "Practical static analysis of JavaScript applications in the presence of frameworks and libraries," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, Aug. 2013, pp. 499–509.

[48] J. Midtgaard and A. Møller, "Quickchecking static analysis properties," *Softw. Test., Verification Rel.*, vol. 27, no. 6, 2017.

[49] F. P. Miller, A. F. Vandome, and J. McBrewster, *Levenshtein Distance: Information Theory, Computer Science, String (Computer Science), String Metric, Damerau? Levenshtein Distance, Spell Checker, Hamming Distance*. Orlando, FL, USA: Alpha Press, 2009.

[50] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient JavaScript mutation testing," in *Proc. IEEE 6th Int. Conf. Softw. Test., Verification Validation*, Mar. 2013, pp. 74–83.

[51] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 2, pp. 158–191, Apr. 1998.

[52] B. B. Nielsen, M. T. Torp, and A. Møller, "Modular call graph construction for security scanning of Node.js applications," in *Proc. 30th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, New York, NY, USA, 2021, pp. 29–41, doi: 10.1145/3460319.3464836.
[53] M. Pradel, P. Schuh, and K. Sen, "TypeDevil: Dynamic type inconsistency analysis for JavaScript," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1. Piscataway, NJ, USA, May 2015, pp. 314–324.
[54] A. Rao and S. J. Steiner, "Debugging from a call graph," U.S. Patent 8 359 584, Dec. 2009.
[55] S. Rasheed and J. Dietrich, "A hybrid analysis to detect Java serialisation vulnerabilities," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, New York, NY, USA, Sep. 2020, pp. 1209–1213, doi: 10.1145/3324884.3418931.
[56] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "PyCG: Practical call graph generation in Python," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, May 2021, pp. 1646–1657, doi: 10.1109/ICSE43902.2021.00146.
[57] S. E. Sim, S. Easterbrook, and R. C. Holt, "Using benchmarking to advance research: A challenge to software engineering," in *Proc. 25th Int. Conf. Softw. Eng.*, May 2003, pp. 74–83.
[58] H. Sun, D. Bonetta, C. Humer, and W. Binder, "Efficient dynamic analysis for Node.js," in *Proc. 27th Int. Conf. Compiler Construct.*, 2018, pp. 196–206.
[59] S. Wei and B. G. Ryder, "Practical blended taint analysis for JavaScript," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2013, pp. 336–346.
[60] T. Xie and D. Notkin, "An empirical study of Java dynamic call graph extractors," Univ. Washington, Seattle, WA, USA, CSE Tech. Rep., 02-12 3, 2002.

GÁBOR ANTAL received the Ph.D. degree in computer science from the University of Szeged, in 2022. He is currently an Assistant Research Fellow with the Department of Software Engineering, University of Szeged. He often develops new tools and methodologies in education to ease the work of both students and teachers. Besides teaching and research activities, he works as a project manager and lead developer on various research and development projects. More than ten publications can be associated with him, with over 140 citations. His research interests include static analysis of dynamic languages, deep learning applications, vulnerability and bug detection, and source code transformation based on abstract syntax trees.

PÉTER HEGEDŰS received the Ph.D. degree in computer science from the University of Szeged, in 2015.

He is currently an Assistant Professor with the Department of Software Engineering, University of Szeged, and a Researcher with FrontEndART Ltd. Besides teaching and research involvement, he also takes part in various software development projects as a project manager and lead developer. His publication record consists of more than 50 papers that appeared in top conferences and high-impact journals. His research interests include software maintainability models, deep learning applications, source code analysis, and vulnerability detection and prediction. He was a PC Member of the CSMR, MSR, ICCSA, and SQM conferences, and currently holds a Bolyai János Research Scholarship.

ZOLTÁN HERCZEG was born in Szeged, Hungary, in 1982. He received the master's degree in computer science from the University of Szeged, Szeged, in 2005.

From 2001 to 2005, he was a Software Developer with Trend Ltd. Since 2005, he has been a Software Developer with the University of Szeged.

GÁBOR LÓKI received the M.Sc. degree, in 2002, and pursued the Ph.D. degree, in 2017. He is currently a Research Assistant with the Department of Software Engineering, University of Szeged. He has been involved in many research and development projects since his graduation, first as a software developer and a research assistant. He is also a technical leader in projects with national and international partners from both the industrial and academic sectors. He is an active researcher, having authored 13 publications. His main research interests include compilers, embedded development, optimizations, dynamic analysis, and benchmarking.

RUDOLF FERENC received the Ph.D. degree in computer science from the University of Szeged, in 2005, and the Habilitation degree, in 2015.

He is currently an Associate Professor and acting as the Head of the Department of Software Engineering, University of Szeged. He leads the Static Code Analysis Group, which develops tools for analyzing the source code of various languages. These tools calculate code metrics and detect coding issues and duplications. He is also leading several research and development projects, which are related to quality assessment, improvement, and architecture reconstruction of software systems for major banks and software development companies in Hungary. His research interests include static code analysis, metrics, quality assurance, design pattern and antipattern mining, and bug detection. He has more than 100 publications in these fields with over 2000 citations. He has been serving as the Program Co-Chair and a Program Committee Member at major conferences, such as ICSE, ICSME, ESEC/FSE, SANER, CSMR, WCRE, ICPC, SCAM, and FASE, since 2005.