An Extensible Approach for Taming the Challenges of JavaScript Dead Code Elimination

Niels Groot Obbink*, Ivano Malavolta*, Gian Luca Scoccia†, Patricia Lago*
*Vrije Universiteit Amsterdam, The Netherlands - niels@nielsgrootobbink.com, {i.malavolta | p.lago}@vu.nl
†Gran Sasso Science Institute, L'Aquila, Italy - gianluca.scoccia@gssi.it

***Abstract*—JavaScript is becoming the de-facto programming language of the Web. Large-scale web applications (web apps) written in Javascript are commonplace nowadays, with big technology players (e.g., Google, Facebook) using it in their core flagship products. Today, it is common practice to reuse existing JavaScript code, usually in the form of third-party libraries and frameworks. If on one side this practice helps in speeding up development time, on the other side it comes with the risk of bringing dead code, i.e., JavaScript code which is *never executed, but still downloaded from the network and parsed in the browser*. This overhead can negatively impact the overall performance and energy consumption of the web app.**

In this paper we present *Lacuna*, an approach for JavaScript dead code elimination, where existing JavaScript analysis techniques are applied in combination. The proposed approach supports both static and dynamic analyses, it is extensible, and independent of the specificities of the used JavaScript analysis techniques. *Lacuna* can be applied to any JavaScript code base, without imposing any constraints to the developer, e.g., on her coding style or on the use of some specific JavaScript feature (e.g., modules).

***Lacuna* has been evaluated on a suite of 29 publicly-available web apps, composed of 15,946 JavaScript functions, and built with different JavaScript frameworks (e.g., Angular, Vue.js, jQuery). Despite being a prototype, *Lacuna* obtained promising results in terms of analysis execution time and precision.**

I. Introduction

Web application frontends are built using a combination of HTML, CSS, and JavaScript. In the last years, the browser has evolved into a fully-fledged software platform (e.g., with JavaScript APIs for geolocation, camera, microphone) [1]. This evolution is leading to a proliferation of third-party libraries and frameworks, ranging from Model-View-Controller (MVC) frameworks, efficient DOM manipulators, UI kits, etc. [2][3]. Today they are even used when developing mobile apps [4, 5]. This is confirmed by the emergence of even websites dedicated to cataloging the large amount of JavaScript frameworks and libraries available to developers [6].

Frameworks increase developers productivity via code reuse and are commonly well tested and maintained. However, they also come with a cost: not every single functionality of a JavaScript framework is usually used by a web app including it. The unused functionalities of the included frameworks are never executed at run-time; the code implementing those unused functionalities is known as **dead code** [7]. Besides the obvious cost of increased file size (and network transfer time), there is an additional hidden cost to dead code: despite JavaScript dead code never being executed at run-time, it is still parsed by the JavaScript engine. This parsing overhead can take a significant portion of the complete interpretation time [8]. The costs for downloading and parsing dead code can negatively contribute to the loading time and energy consumption of web apps. A recent survey among almost 9,300 JavaScript developers rated *code splitting* and *dead code elimination* as the highest-rated requested features [9].

While some approaches have been developed to minimize this overhead (e.g., lazy parsing [10] and script streaming [11]), dead code identification and elimination is still an open problem. Currently available solutions either (i) impose a certain coding style to developers, banning certain code structures (e.g., object reflection), or (ii) require specific constructs of the JavaScript specification. An example of the latter is the use of *modules*, which allow developers to declaratively specify self-contained namespaces in JavaScript and to conditionally load them when needed. While modules are certainly useful in terms of maintainability and code reuse, most web apps today have not been built with modules in mind. Also, at the time of writing, only Google Chrome and Safari support JavaScript modules[1].

In this paper we present *Lacuna*, an approach for automatically eliminating JavaScript dead code from web apps. At the core of *Lacuna* lies the construction of a call graph G_w of the web app w being analysed; G_w is uni-directed and represents JavaScript functions as nodes and the caller-callee relationship between functions as edges. In this context, dead code elimination consists in the removal of all the (potentially connected) components in G_w that are isolated from the node representing the global scope of the web app. The unique characteristic of *Lacuna* is its ability to *build and iteratively refine* G_w by integrating different program analysis techniques, each of them with potentially its own support for specific aspects of the JavaScript language. *Lacuna* supports any kind of program analyses (both static and dynamic), provided that they are aimed at building a call graph of the JavaScript code being analysed. *Lacuna* iteratively refines its own representation of G_w by suitably integrating it with the call graph produced by each analysis technique. *Lacuna* is *extensible* and independent from the used program analysis techniques, allowing developers and researchers to build the combination of analyses which best fits their own needs. Finally, *Lacuna* can be applied to any JavaScript-based web app, without

[1] http://caniuse.com/#feat=es6-module

978-1-5386-4969-5/18/$31.00 © 2018 IEEE

imposing any constraints to the developer, e.g., on coding style (use of reflection or objects self-inspection) or imposing the use of specific JavaScript features (e.g., modules).

The main **contributions** of this paper are the following:

- the definition of an extensible approach for JavaScript dead code elimination, without imposing restrictions on the coding style of developers;
- an implementation of the proposed approach in Node.js;
- three extensions of the approach leveraging existing JavaScript analyzers;
- an evaluation of the proposed approach based on a publicly-available dataset of 29 web apps;
- a complete replication package containing the web apps used in the evaluation, raw data, and analysis scripts.

The **target audience** of this paper is web app developers and researchers. Web developers are the natural stakeholders of *Lacuna* since they can directly use the current implementation of *Lacuna* for removing dead code from their web apps, thus making their products more lightweight in terms of, e.g., network usage, loading time, energy consumption. Researchers can use *Lacuna* as a means for assessing their analysis techniques for JavaScript dead code elimination, and for quickly comparing the performance of their techniques with respect to other analyses already integrated in *Lacuna*.

In the remainder of this paper, Section II presents basic concepts and the challenges of JavaScript dead code elimination. Section III presents the proposed approach and Section IV gives implementation details. Section V describes the empirical evaluation of our approach, and in Section VI we discuss related work. Section VII closes the paper.

II. Background

In this section we set the context of this paper by discussing the main challenges related to the analysis of JavaScript code (Section II-A), and existing tools and techniques.

A. The Challenges of JavaScript Analysis

Due to the dynamic nature of JavaScript, it is hard to completely and correctly analyze its source code [12]. The following section highlights several language features that are challenging from the perspective of JavaScript analysis tools, with a special emphasis on those issues which make call graph analysis for JavaScript especially difficult.

1) Dynamic access of objects properties: In JavaScript it is possible to dynamically access the properties of objects. This is a particularly hard problem for static analyzers, because it requires the interpretation of the run-time context of the code being analysed [13].

```
1 function get (mom, unit) {
2     return mom.isValid() ?
3         mom._d['get' + (mom._isUTC ? 'UTC' : '')
4             + unit]() : NaN;
5 }
```

Listing 1. Dynamic property access (from the Chart.js framework)

As an example, Listing 1 shows a fragment of a existing JavaScript framework (Chart.js[2]). Here, the `get` function defined in line 1 takes as input two variables (i.e., `mom` and `unit`) and returns the value of `mom` if it is valid (line 2); if `mom` is valid (line 3), a property in `mom._d` (a `Date` object) is dynamically accessed by name, depending on the value of `mom._isUTC` property (i.e. `'getUTCMonth'` or `'getMonth'`). In this case the problem is due to the fact that the name of the function to be called (e.g., `getUTCMonth()`) is crafted at run-time and depends on the specific value of `mom._isUTC` at run-time.

Dynamic object properties are challenging for static analysis techniques, because determining what function will be called depends on the value of `mom` at run-time.

2) Context binding: Context binding allows developers to assign an arbitrary object to the `this` keyword value, using the `call`, `apply` or `bind` functions[3]. Dynamic context is often exploited in JavaScript frameworks since it gives great flexibility to developers, e.g., for functional programming or for defining variadic functions (i.e., functions accepting a variable number of arguments).

```
1 _.after = function(times, func) {
2     return function() {
3       if (--times < 1) {
4         return func.apply(this, arguments);
5       }
6     };
7   };
8   var isBlockedUser = _.after(3, function () {
9       console.log('Wrong password for 3 times.');
10    });
```

Listing 2. Context binding (from the underscore.js library)

Listing 2 shows an example from the well-known underscore.js[4] library. Here, the `_` variable represents the root object of the library. The `after` function in lines 1-7 allows developers to obtain a function `func` in such a way that it is executed only on and after the `times`-th call. At the core of the `after` function lies a call to the `apply` function (line 4), where `func` is correctly called independently of its execution context (`this` and the passed arguments). In lines 8-10 of the listing the `after` function is applied on a function which just prints a message in the console after it has been called 3 times.

Similarly to the dynamic access of objects properties, dynamic context binding is a challenge for static analysis tools since it requires the knowledge of both the run-time context and the scope of the object it is binding onto.

3) Run-time input uncertainty: While dynamic analysis does not suffer from the issues presented above, it may suffer from run-time input uncertainty. Specifically, not only do many web apps require user input, but the run-time execution flow is dependent on this input. Login screens or prompting for user input are typical cases of run-time input uncertainty. In this

[2]http://www.chartjs.org

[3]http://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

[4]http://underscorejs.org

context, the challenge is that, while it might be possible for an analyzer to deal with simple events (e.g., check whether a button is clicked), it is difficult to check every possible event combination, especially considering that some events may depend on the user's device (e.g., a tap on a specific coordinate of the display).

4) Eval and the Function constructor: The `eval` function and `Function` constructor take a string as input and execute it as JavaScript code at run-time. Albeit some progress has been made on analyzing [14] and automatically transforming [15] these constructs, their usage is still hard to analyze, mainly because the code string can be dynamically built.

5) Dynamic script injection: Due to the ability for JavaScript to access the DOM, it is possible to dynamically inject and execute `script` tags into the DOM in order to programmatically control what to execute and when. Listing 3 shows an example of dynamic script injection, where an empty `script` tag is created (line 1), linked to an external JavaScript file (line 2), and finally injected into the `head` of the DOM.

```
1 var stNode = document.createElement("script");
2 sNode.src = "path-to-script.js";
3 document.head.appendChild(sNode);
```

Listing 3. Example of dynamic script injection

JavaScript libraries, such as the widely-used *requireJS*, use this technique to load script files dynamically, ensuring they are loaded in the order provided by the developer. Since in those cases the injected JavaScript code is dynamically loaded, program analysis techniques can struggle with its correct identification.

6) Timers: JavaScript provides APIs for setting timeouts or for repeatedly calling functions over time (e.g., the `setTimeout` and `setInterval` functions). These functions are part of the standard APIs provided by modern browsers and may be a challenge for both static and dynamic analyses. If we consider a web app that triggers a function to display a popup (e.g., for showing a subscribe to a newsletter lightbox) after several minutes of browsing, a dynamic analyzer would have to wait for the timer to be triggered for correctly identifying the function called by the timer.

B. Existing Dead Code Elimination Tools

rollup.js is an ES6 module bundler that excludes unused modules and performs dead code elimination using a process known as *tree-shaking* [16]. Similarly, Webpack (an asset bundler) also supports tree-shaking [17]. This is an effective way of (partial) dead code elimination, but requires the developer to use ES6 modules which are not widely supported at the time of writing [18]. Another issue is that it requires the developer to meticulously write import and export statements, because otherwise unused functions might still be imported. For libraries and frameworks, this makes users dependent on the authors to provide correct export calls.

The *Google Closure Compiler* is a tool that aims to make JavaScript code run faster. It analyses the source code, removes dead code and rewrites it to a more optimal form [19]. It supports multiple compilation levels, ranging from minification to complete code transformations. While the Closure Compiler has some impressive results for dead code elimination, it also has a drawback. The simple optimization level does not remove unused functions at all. The advanced optimization level does remove unused functions, but requires a specific coding style[5].

C. Building Call Graphs for JavaScript

At the core of *Lacuna* lies the construction of the call graph of the JavaScript code of the web app under analysis. A call graph is a special kind of control-flow graph where each node represents a function and each edge represents a function call [20]. The complete call graph of a program might contain nodes that are never reached (i.e., functions that are never called at run-time). As an example, consider Listing 4, the functions `a` and `b` are reachable, but `c` is not.

```
1 function a() {
2   b();
3 }
4 function b() {
5   b();
6 }
7 // dead function
8 function c() {
9   a();
10 }
11 a();
```

Listing 4. Example of dead function (`c`)

A visual representation of the call graph for the listing above is shown in Figure 1. Traversing this graph from the starting node (in the case of JavaScript, the *global* scope), and marking the visited nodes, will result in a list of all functions that are called when executing the code in the listing. In this example, when performing dead code elimination, function `c` would be removed, as it is unreachable when traversing the call graph. This removal operation will save the time and resources used by the browser for downloading and parsing the `c` function.

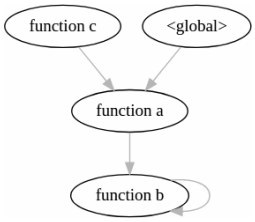

Fig. 1. Visual representation of the call graph of Listing 4

Unfortunately, due to the challenges introduced in Section II-A, finding a correct and complete call graph is extremely difficult today. The intuition behind *Lacuna* is that combining different types of analyses can lead to more precise results. For instance, Listing 5 contains a long timeout definition (line 5) and dynamic context binding (lines 12-14), which are challenging for dynamic and static analysis techniques, respectively.

[5]https://developers.google.com/closure/compiler/docs/limitations

```
var object = {
  property: "default",
  method: function() {
    var prop = this.property;
    setTimeout(f, 1000 * 60 * 5);
  }
};
function f(){
  console.log(prop);
}
// dynamic context binding
object['method'].apply({
  property: "overwritten"
});
```

Listing 5. Example of code with issues for different analysis types

In this case, the isolated execution of static and dynamic analyses would report only an incomplete call graph. Specifically, dynamic analysis would not identify the execution of the function `f`. Instead, static analysis would not identify the execution of the function in the `obj.method` because the inner function is not reachable from the global scope. However, when combining the (partial) call graphs of the two analyses, it is possible to obtain a complete call graph since each type of analysis contributes to different parts of the overall call graph.

D. Existing JavaScript Analysis Tools

Currently several tools for analyzing and inspecting JavaScript source code are available. In the following we give an overview of the most used JavaScript source code analysis tools. JSAI is a static JavaScript analyzer. It constructs an intermediate form, known as *notJS*, based on the abstract syntax tree of the web app being analysed [21].

A similar tool is TAJS, which statically infers type information and call graphs for JavaScript [22]. An implementation written in Java is available[6].

WALA is a static analysis framework for Java and JavaScript [23]. Similarly to JSAI, WALA builds an intermediate form of the JavaScript code being analysed. WALA allows for pointer analysis and call graph construction.

JSFlow is a JavaScript interpreter for tracking information flow and is written in JavaScript [3].

Esprima[7] is a JavaScript parser written in JavaScript. It produces a syntax tree for a given JavaScript file, which can be easily traversed. While Esprima does not provide any analysis on its own, it is a useful tool for retrieving lexical information of the JavaScript code of a given web app (e.g., for retrieving function definition locations).

A static analyzer has been developed based on an approach utilizing point analysis [24]. It makes use of Esprima, and builds an approximate call graph, ignoring dynamic properties and context binding.

It is important to note that our approach can take advantage of these tools. Indeed, instead of competing with them, we aim at combining multiple analyzers in order to leverage their strengths and complement their weaknesses.

[6]https://github.com/cs-au-dk/TAJS

[7]http://esprima.org

III. The Approach

In this section we use a purposefully-simple web app as running example for describing each phase of *Lacuna*. The HTML code of the web app just includes the `example.js` script in Listing 6. it defines the `a`, `b`, and `c` functions (lines 1-14) and calls `a` in line 15. When called, function `a` starts a 6-seconds timeout (lines 2-4), and function `b` prints a message in the browser console when the timeout is triggered (lines 3 and 7). In this example, functions *a* and *b* are alive, whereas function *c* is dead code. As shown in Figure 2, *Lacuna* is composed of three main phases, namely: *Parsing*, *Analysis*, and *Elimination*.

```
function a(){
    setTimeout(function(){
        b();
    }, 6000);
}
function b(){
    console.log('6 seconds have passed');
}
function c(){
    console.log('function c has been called');
    /*
    Other potentially heavy statements
    */
}
a.call();
```

Listing 6. Running example - `example.js`

A. Parsing

This phase takes as input the source code of the web app being analyzed (*w* in Figure 2). It is important to note that *w* is the only input needed by *Lacuna*, making it applicable in the context of a wide spectrum of projects, independently of the used development process or company-specific practices.

Firstly (step (1) in the figure), *Lacuna* identifies all the JavaScript code within *w* by including (i) all the JavaScript code defined in-line in the HTML code in *w*, (ii) all JavaScript files referenced by the HTML code in *w* by means of the `<script>` tag, and (iii) all the JavaScript files in *w*. Once all the JavaScript code related to *w* has been identified, *Lacuna* parses it into an internal representation of all its statements in order to ease subsequent steps. This part is realized using the Esprima parser.

In step (2), all function definitions within *w* are retrieved (including anonymous and inline functions), and we instantiate an initial call graph G_0 containing a node for each identified function declaration. Additionally, a starting node representing the JavaScript global scope is included in G_0 in order to be able to consider also all those functions directly called from the global scope of the web app.

The G_0 call graph of our running example contains five nodes, namely: the *global* node, one node for each *a*, *b*, and *c* functions, and one node for the inline function defined in the *setTimeout* call. G_0 does not contain any edge in this phase, they will be considered in the next phase.

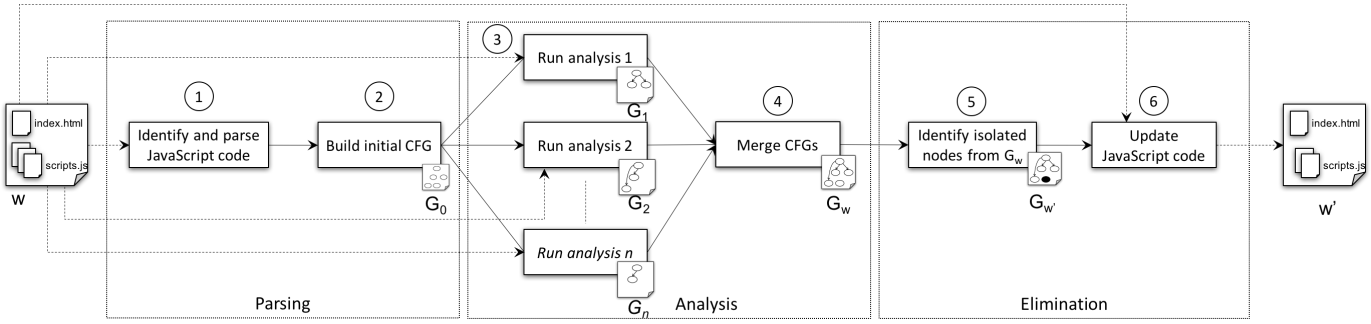

Fig. 2. Overview of our approach.

B. Analysis

By taking as input the G_0 call graph, this phase will produce as output a call graph G_w with the same nodes as G_0 and the edges representing the caller-callee relationship between the JavaScript functions of w. Specifically, an edge e_{ij} between the node i and the node j in G_w represents the fact that the JavaScript function i is able to call the JavaScript function j in w. Due to the highly dynamic and event-based nature of Javascript (see Section II), the identification of the e_{ij} edges in G_w is difficult. As of today there is no technique for building correct and complete call graphs for JavaScript without imposing any constraints to developers or making strong assumptions on the usage of the language, e.g., having a complete test suite or prohibiting the use of reflection.

The key idea of this phase is to embrace this inherent limit of the JavaScript language, and to **include** different analysis techniques and **merge** their corresponding outputs. This allows us to leverage the strengths of existing analysis techniques, while potentially mitigating the issues they may have with certain constructs of JavaScript. *Lacuna* is independent of the internal assumptions and techniques applied in each included analysis technique and can execute all of them in parallel. It is important to note that an included analysis technique x can be of any kind, either dynamic or static, since *Lacuna* executes it as a black-box component. Here, the only assumption is that each analysis technique x adheres to the interfaces provided by *Lacuna*, meaning that it has to take as input G_0 and the source code of w (step (3)), and builds its own call graph G_x. In Section IV we will show that it is relatively straightforward to wrap already existing analysis tools (e.g., WALA [23]) in order to include them into *Lacuna*.

Step (4) merges the call graph G_x produced by each analysis technique x into a final call graph G_w. G_w has the same definition of G_0: a node for each JavaScript function in w and an edge e_{ij} for each call to j from the body of a function i in w. In addition, in G_w edges are labelled with the identifier of the analysis technique which discovered it. In order to take into account the fact that multiple analysis techniques can identify the same function call as executed, each edge in G_w can have multiple labels.

More in details, the strategy we apply in step (4) is the following: (i) each node in G_0 is replicated into G_w, (ii) we iterate over all the call graphs $G_1 \dots G_n$ produced by the n analysis techniques included in *Lacuna* and for each G_x in $G_1 \dots G_n$ we add all its edges into G_w, (iii) when adding an edge e_{ij} produced by a technique x, if e_{ij} is already in G_w, then we just add the label x to e_{ij}.

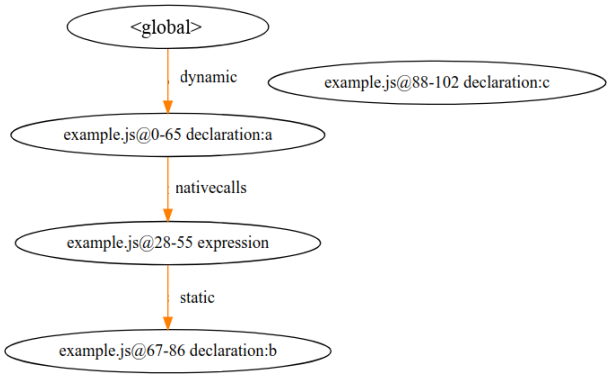

Fig. 3. Running example - G_w. Format of node labels: `sourcefile@s-e type[:name]`, where `s` and `e` are the start and end character positions of the function definition in `sourcefile`, `type` can be either *declaration* or *expression*, and `name` is its name.

Figure 3 shows the merged call graph G_w of our running example after running the three analysis techniques included in the current implementation of *Lacuna* (they will be explained in detail in Section IV), namely: *static*, *dynamic* and *nativecalls*. Here, the *dynamic* analysis identified the call from the global scope to the function *a*, *nativecalls* identified the call from *a* to the inline function defined in lines 2-4 in Listing 5 (by considering the call to *setTimeout* as a direct function call), and the *static* analysis identified the call from the body of the previous function definition to *b*. No analysis technique identified any call to function *c*, so it is unreachable from *global* because it has no incoming edges at all.Interestingly, we can notice that each analysis technique contributes only partially to the overall call graph G_w. Indeed, when used in isolation, they could have brought to potentially incomplete call graphs (e.g., the calls to the inline function and *b* have not been identified by *dynamic* analysis alone). In *Lacuna* we aim at leveraging the complementarity of existing program analysis techniques for JavaScript in order to build a call graph as precise as possible from the dead code elimination perspective.

C. Elimination

Once all analysis techniques have been executed and the complete G_w is available, the elimination phase identifies all the nodes in G_w representing dead code (step (5)). We consider dead code each node i in $G_w - \{global\}$ which is unreachable from the *global* node (i.e., there is no sequence of adjacent nodes starting from *global* and ending to i). These nodes represent JavaScript functions that are not called by any other function (or from the global scope) according to all the different analysis techniques applied in the previous phase. We consider the global node as always alive since in JavaScript the global scope is always present and executed in a web app running in the browser[8]. By referring to Figure 3, in our running example the only node which is (i) different from the global one and (ii) unreachable from the global node is the one corresponding to the *c* function.

In step (6), *Lacuna* reconsiders the source code of the input web app w and removes the JavaScript functions corresponding to isolated nodes in G_w. In the current implementation of *Lacuna* we perform this operation by removing the body of each JavaScript function to be removed; the rationale for this choice is that in many cases function declarations are used as expressions in JavaScript and are used in various contexts in which just removing the function declaration would end up in run-time errors in the browser. For example, by referring to Listing 6, removing the function declaration in line 2 could have resulted in a run-time error when calling the setTimeout() function, which is expecting exactly two parameters (the first one for the function to be called, and the second one for the delay in milliseconds).

After the execution of step (6), the dead-code-free w' produced by *Lacuna* is the same JavaScript code in Listing 6, where the body of the *c* function has been removed completely.

IV. Implementation

We developed *Lacuna* as a Node.js application composed of 30,474 lines of JavaScript code. The *Lacuna* implementation can be run as a command-line tool, making it easy to include into a continuous integration/build system. It takes as input the path of the web app to be analyzed and makes an in-place update of its source code files by removing the identified JavaScript dead code. The *Lacuna* tool is also able to (visually) output both G_w and G'_w call graphs. An example of a call graph produced by *Lacuna* is shown in Figure 4. Edges are marked with the name of the analysis technique that detected them, and have a progressive colour (red to green) based on the number of analysis techniques identifying them.

A. Realized Extensions

The following gives an overview of the analysis techniques we already included in the current implementation of *Lacuna*. We developed those extensions with two purposes in mind: (i) to provide examples about how to extend *Lacuna* and (ii) to use them in empirically evaluation of *Lacuna* (see Section V).

[8]http://www.w3schools.com/js/js_function_invocation.asp

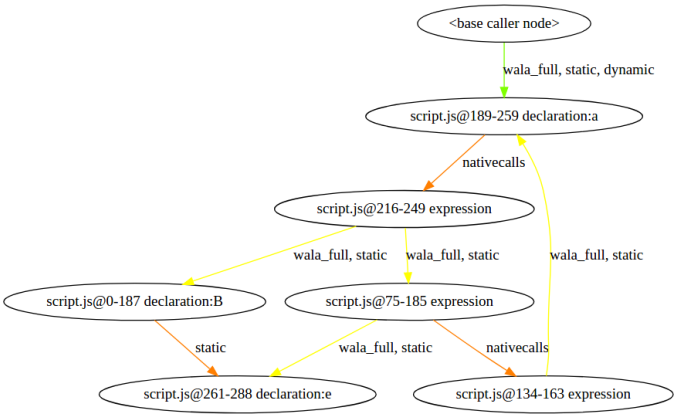

Fig. 4. Example Lacuna function graph output for a trivial JavaScript file.

1) Dynamic analyzer: We implemented a minimalistic dynamic analyzer for web apps. Firstly, it instruments w by adding logging statements at the beginning of the body of every function definition in w (including anonymous and inline functions). Then, it runs the web app in a headless browser (namely, PhantomJS[9]), collects the logging information, and builds the call graph according to the functions executed at run-time.

2) Static analyzer: We implemented a basic static analyzer which relies on the implementation[10] of the *field-based call graph construction tool* discussed in Section II-D. In addition, we extended such implementation by considering also native JavaScript functions (i.e., `Array.prototype.map` or `Function.prototype.call`) when building the call graph.

3) WALA-based extensions: We wrapped the well-known WALA analysis tool and produced two different extensions: *wala_single* and *wala_full*. They both use WALA as backend; *wala_single* analyzes JavaScript files one by one, whereas *wala_full* considers the web app as a whole. *wala_single* has a relatively short execution time because it considers each JavaScript file in isolation, whereas *wala_full* considers function calls across different JavaScript files.

B. Assumptions and Limitations

Due to the prototypal stage of *Lacuna*, we made the following assumptions when implementing it.

Like most existing JavaScript analysis tools, our implementation of *Lacuna* assumes that the web app w is correct, i.e., it does not produce errors at run-time, and it does not contain broken links or missing functions. This is not always the case in real-world web apps. Introducing error handling for these cases would greatly increase the stability of the current implementation of *Lacuna* and allow us to perform a more in-depth experimentation with real-world web apps. This activity is already planned as part of future work.

[9]http://phantomjs.org
[10]http://github.com/abort/javascript-call-graph

The extension for dynamic analysis is quite minimalistic. While this has its merits, it is lacking some more thorough analysis, as it cannot cope with all JavaScript events (e.g. long setTimeout functions). Also, it neglects events coming from user interaction (e.g., clicking on a button), which in turn may trigger the execution of JavaScript functions listening to them.

Our WALA-based extensions inherit one of the most known limitations of WALA: they require a very long time for completing the analysis [25]. Analysis time with WALA greatly increases with the size and number of functions in JavaScript files in w. This means that those extensions cannot be used for arbitrary applications within reasonable time.

Our current implementation assumes that w has only one initial HTML page (its `index.html` file). Of course, w may contain multiple HTML pages, but they are currently discarded by *Lacuna*. We can refine the parsing phase of *Lacuna* to search and analyze all HTML files in w, this assumption has been made due to time/resource constraints only.

V. Evaluation

In this section we report the design and the results of an experiment we performed for evaluating the current implementation of *Lacuna*. To allow easy replication and verification of the experiment, we provide interested researchers a complete **replication package**[11]. It includes the source code of the current implementation of *Lacuna*, the source code of the web apps we consider as subjects of the experiment (before and after dead code removal), the raw data of the experiment, and the R scripts for analysing obtained results.

A. Experiment Design

The **goal** of this experiment is to empirically evaluate the following aspects of *Lacuna*: (i) its overall *performance* in terms of execution time and (ii) its *accuracy* in identifying and removing JavaScript dead code.

In order to achieve this goal in an objective and replicable manner we planned our experiment by following well-known guidelines on empirical software engineering [**?** 26]. In the following we report how we designed the experiment.

Research questions. The above-mentioned goal has been refined into the following research questions:

RQ1 - *What is the execution time of Lacuna when performing different types of analysis techniques?*

RQ2 - *What is the accuracy of Lacuna when performing different types of analysis techniques?*

To answer these questions, we executed the current implementation of *Lacuna* on a set of web apps developed with different JavaScript frameworks. Since *Lacuna* can apply different types of analysis techniques, it is expected that combining the analyses will result in higher execution times and, more importantly, in an improvement in terms of accuracy in JavaScript dead code elimination.

Subjects selection. For this experiment we considered 29 web apps belonging to the TodoMVC[12] project. The TodoMVC project hosts different implementations of the same web app, where different JavaScript frameworks and coding practices are applied. The realized app is a simple to-do list manager, which provides the following features to the user: (i) to add a todo item, (ii) to remove a todo item, (iii) to edit an already existing todo item, and (iii) to mark a todo item as completed.

The main goal of the TodoMVC project is to enable practitioners to study and compare MV* (Model-View-Anything) JavaScript frameworks through source-code inspection of the same web app, developed by experienced web developers employing their favourite JavaScript frameworks [27].

TABLE I
SUBJECTS OF THE EXPERIMENT

Web app	All functions	Alive	Dead	Best F-score
ampersand	774	385	389	0.68
angularjs-require	127	71	56	0.63
backbone	950	400	550	0.73
canjs	1105	577	528	0.68
canjs-require	128	67	61	0.59
closure	591	291	300	0.66
dijon	834	356	478	0.74
dojo	1074	591	483	0.60
elm	965	310	655	0.82
enyo-backbone	20	7	13	0.77
exoskeleton	256	176	80	0.46
gwt	45	21	24	0.65
jquery	869	376	493	0.73
js-of-ocaml	1352	366	986	0.86
jsblocks	885	433	452	0.68
knockoutjs	560	315	245	0.58
knockoutjs-require	128	67	61	0.57
mithril	136	70	66	0.74
olives	533	227	306	0.75
puremvc	229	143	86	0.55
ractive	932	569	363	0.60
serenadejs	400	254	146	0.53
somajs	342	208	134	0.58
somajs-require	128	71	57	0.58
spine	999	456	543	0.69
troopjs-require	130	71	59	0.54
typescript-backbone	1243	455	788	0.75
vanilla-es6	80	53	27	0.56
vanillajs	131	107	24	0.37
Mean	549.8	258.3	291.4	0.64

We selected the web apps provided in the TodoMVC project as subjects of our study because: (i) they can be considered as real-world web modern apps, (ii) they make use of a highly heterogenous set of JavaScript frameworks (e.g. Backbone.js, jQuery, Angular, Bootstrap), and (iii) their relatively small size (both in terms of provided features and source code) allowed us to straightforwardly execute and profile them during the experiment. The web apps in the TodoMVC project can be considered as representative of real-world web apps since they (i) include many common JavaScript coding patterns (e.g., event handlers, local data storage, internal state management, front-end rendering), (ii) similar to what happens in many software companies, they must comply with a rigorous set of requirements, HTML/CSS templates, coding styles, and other specifications [27], and (iii) their code is of high quality since they are included in the TodoMVC catalog only if they pass a first review by the TodoMVC project leaders and a second review by the open-source community [27].

[11] http://github.com/NielsGrootObbink?tab=repositories

[12] http://todomvc.com

In Table I we report the 29 web apps we considered in this experiment (in the table each web app is identified by a mnemonic name of the main JavaScript frameworks it uses). In order to keep the execution of the experiment under control, we excluded those web apps in the TodoMVC catalog that could not be served in our own web servers (e.g., those depending on external services like Firebase[13]), and those developed using languages different from JavaScript (e.g., React, Dart, Scala.js).

As can be seen in the *All functions* column of the table, our dataset is quite heterogenous in terms of number of JavaScript functions, with a range between 20 (*enyo-backbone*) 1352 functions (*js-of-ocaml*), average=549.9 (median=553), standard deviation=421.78, and coefficient of variation=76.7%.

Variables. In this experiment, the only independent variable represents the type of analysis technique used for building the call graphs. This variable is used for answering both RQ1 and RQ2 and can have three treatments: *static*, *dynamic*, and *hybrid*. The first two treatments correspond to the static and dynamic analyzers discussed in Section IV, whereas the latter represents their usage in combination (i.e., static analysis + dynamic analysis). We also explored the possibility of using our WALA-based extension, but its extremely long execution time prevented us from using it in this experiment [25].

Concerning RQ1, the dependent variable is the execution time in milliseconds of the current implementation of *Lacuna*. These values are directly produced by *Lacuna* at each run. Concerning RQ2, we borrow the precision and recall metrics from the field of information retrieval [28]. In our experiment, the precision P_a of an analysis technique a when analyzing a web app w is defined as:

$$P_a(w) = \frac{|dead_a(w) \cap dead_t(w)|}{|dead_a(w)|} \quad (1)$$

Where $dead_a(w)$ is the set of functions identified as dead by the analysis technique a and $dead_t(w)$ is the set of true dead functions in w. Intuitively, precision deals with false positives, it can be seen as the fraction of JavaScript functions identified as dead by the analysis a that are true dead functions.

Differently, the recall R_a of an analysis technique a when analyzing a web app w is defined as:

$$R_a(w) = \frac{|dead_a(w) \cap dead_t(w)|}{|dead_t(w)|} \quad (2)$$

Recall deals with false negatives, and it is defined as the fraction of JavaScript functions identified as dead by the current analysis, over the total amount of true dead functions contained in the web app. In order to have a single measure for representing the accuracy of *Lacuna*, we also use the $F-score$, which is a combined metric defined as the harmonic mean of precision and recall [28].

Experiment execution. Firstly, we obtained ground truth values by computing true dead and alive functions of each web app in our dataset. Specifically, for each of the 29 web apps in our dataset, we firstly obtained its total number of JavaScript functions by analyzing its source code with the Esprima JavaScript parser and counting all the occurrences of function definition expressions. Then, we identified true alive functions by (i) programmatically inserting log calls in every JavaScript function identified in the previous step, (ii) executing all the features of the web app by manually using all its functionalities (this is possible since in the TodoMVC catalog, apps functionalities are always the same and limited in number), and (iii) collecting the logged function calls at the end of the manual execution. Finally, we considered as true dead functions all those functions that have not been identified as true alive functions. The last two columns of Table I show the true alive and dead functions of the web apps considered in this experiment, respectively.

Secondly, for each type of analysis (i.e., static, dynamic, and hybrid), we performed the following steps: (i) we ran the *Lacuna* tool on each web app in our dataset, (ii) we extracted its execution time from the logs produced by the *Lacuna* tool itself, and (iii) we computed its precision and recall by applying formulas (1) and (2) on the results produced by *Lacuna* and the baseline values.

The whole experiment has been performed on a Linux laptop with 8GB RAM, an Intel Core i7-4710HQ Processor, and Node.js version 7.5.0.

B. Experiment Results

1) Execution time (RQ1): The average execution time of *Lacuna* is 51.19 seconds (median = 60.95), with a minimum of 0.3 seconds for *enyo_backbone* and a maximum of 159.39 seconds for *canjs*. The main reason for this difference lies in the different sizes of the considered web apps, where *enyo_backbone* is composed of 20 JavaScript functions, whereas *canjs* is composed of 1105 functions. We noticed this trend for all the types of analysis (i.e., static, dynamic, and hybrid).

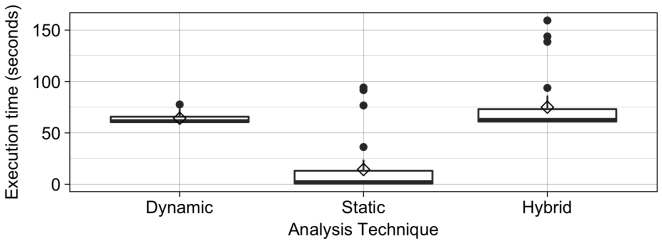

Fig. 5. Execution time of *Lacuna* depending on the used analysis techniques (black bar = median, diamond = average).

Figure 5 shows the breakdown of the execution times of *Lacuna*, depending on the applied type of analysis technique. The type of performed analysis is clearly impacting the execution time of *Lacuna*. Indeed, the execution time of *Lacuna* when applying dynamic analysis (average=64.23s, median=61.45s, SD=51.11s) only is far higher than its execution time when applying static analysis only (average=14.36s, median=2.17s, SD=26.72s). This difference is mainly due to the fact that the

[13]http://firebase.google.com

dynamic analysis has to load the whole web app, whereas the static analysis parses the HTML and JavaScript code directly from their files. Moreover, the execution time of *Lacuna* with static analysis can be considered as an upper bound of the execution time of the parsing and elimination phases of *Lacuna* (see Figure 2); so, the obtained results are also a good indication of the fact that our implementation of the parsing and elimination phases is quite efficient in terms of execution time.

Finally, we can also notice that the execution time of *Lacuna* when performing both static and dynamic analysis (i.e., the *hybrid* case) inherits the execution times of both the types of analyses, i.e., the execution times sum up. This is expected since, even if the approach has been designed for supporting analyses techniques independence, the current implementation of *Lacuna* executes them in sequence. We are planning to support parallelism in the next release.

Results (RQ1). In average, the execution time of *Lacuna* is below 1 minute. It can depend on the size of the input web app w and on the type of used analysis techniques. The parsing and elimination phases of *Lacuna* are quite efficient in terms of execution time.

2) Accuracy (RQ2): The average F-score of *Lacuna* is 0.53 (median = 0.51, SD = 0.14). We recall here that the F-score metric ranges between 0 and 1 and represents the trade-off between precision and recall [28]. This result is quite promising and shows the viability of our approach. In the following we will zoom into the accuracy of each analysis type (i.e., static, dynamic, hybrid).

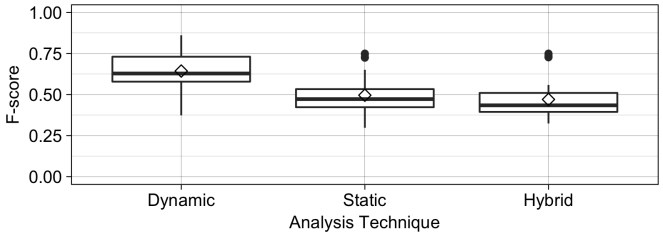

Fig. 6. F-score of *Lacuna* for each analysis technique.

Figure 6 shows the distribution of the **F-score** achieved by *Lacuna* for each analysis technique. The results of the experiment show that *Lacuna* tends to be more accurate when applying dynamic analysis (average = 0.64, median = 0.62, SD = 0.10), rather than when it uses the static (average = 0.49, median = 0.47, SD = 0.12) and hybrid ones (average = 0.47, median = 0.43, SD = 0.12). This result is quite surprising since intuitively one may expect that a combination of the analyses (i.e., the hybrid case) leads to better overall accuracy. In order to better investigate on this result, we analyze the precision and recall of *Lacuna* in isolation. Figure 7 shows its precision and recall for each type of analysis.

Here we can notice that the **precision** of *Lacuna* (red boxplots) is quite high and stable when using different types

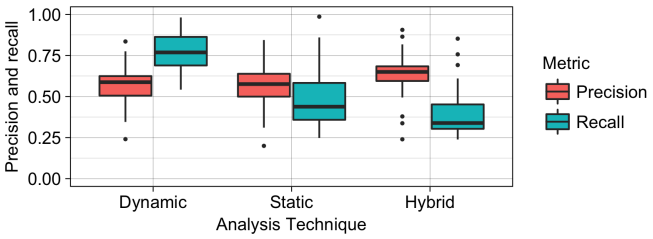

Fig. 7. Precision and Recall of *Lacuna* for each analysis technique.

of analysis. *Lacuna* had an average (median) of 0.57 (0.59), 0.56 (0.58), and 0.63 (0.65) for dynamic, static, and the hybrid analyses, respectively. In this case, the hybrid analysis technique performed even better than the others. This result is expected since static and dynamic analyses build call graphs differently, and in the merging phase *Lacuna* merges them in such a way to minimize the number of functions identified as dead; it may be more probable that the functions still identified as dead after the merge step are true positives.

```
1 Controller.prototype.showActive = function () {
2   var self = this;
3   self.model.read({ completed: false }, function
        (data) {
4     self.view.render('showEntries', data);
5   });
6 };
```

Listing 7. Example of function from the *vanillajs* web app.

As an example, the *vanillajs* is one of the web apps where hybrid analysis performed better in terms of precision. Listing 7 shows a function (lines 3-5), where static analysis failed and dynamic analysis succeeded in determining its liveness. Indeed, the dynamic analysis considered this code as alive because it is executed when the app is loaded in the browser, whereas static analysis failed to identify it because of context binding (`var self = this` in line 2).

When looking at the **recall** of *Lacuna* (green boxplots), we can notice that it ranges from very *high values for dynamic analysis* (average = 0.77, mean = 0.77), to much lower values for static (average = 0.49, mean = 0.44) and hybrid analyses (average = 0.40, mean = 0.34). After a manual inspection of the analysed web apps, the high level of recall for dynamic analysis is mainly due to two main reasons. Firstly, many web apps in our dataset import third-party libraries usually containing a high number of JavaScript functions (e.g., jQuery), but they use a very small part of those libraries; dynamic analysis correctly identified unused functions as dead. Secondly, the business logic of the considered web apps is quite minimal (it has just to manage a list of to-do items); this is reflected in a low number of app-specific JavaScript functions which may not be executed during dynamic analysis. The latter point explains also why the recall of static analysis is low.

Even if the low recall achieved when using hybrid analysis may seem surprising, it is an expected result and it is a direct consequence of how *Lacuna* merges the call graphs of the performed analyses. Specifically, the merging of all the call graphs G_1, G_2, G_3 produced by the 3 analysis techniques is

incremental, meaning that the combined call graph G_w always contains more edges with respect to the single call graphs G_1, G_2, G_3. Having more edges means that potentially less functions are isolated in the final G_w, resulting in a lower number of (potentially alive) dead JavaScript functions.

In other words, our experiment proved that the combination of the analyses techniques improves the precision of *Lacuna* ($\rightarrow$ less false positives), whereas it is detrimental from the recall point of view ($\rightarrow$ more false negatives). As future work, we will extend the merging step so that it will consider the families of dynamic and static analyses in different ways in order to better exploit their specific characteristics. As an example, if an edge e_{ij} is identified during dynamic analysis, we can consider the function j as 100% alive; differently, if e_{ij} is identified during static analysis, then the function j is alive only with a certain degree of confidence.

Results (RQ2). The overall accuracy of *Lacuna* is promising, and there is still room for improvement. Precision is quite high and stable (60%), and it improves when combining analyses techniques. Differently, recall (55%) is very high for dynamic analysis (77%); it drops when considering static analysis (49%) and its combination with dynamic analysis (hybrid, 40%).

VI. Related Work

While there has been considerable interest in static analysis of Javascript programs (specially in the security area), at the time of writing we found no research focused on removal of dead code at the function level from arbitrary web apps.

Guarnieri et al. propose *Gatekeeper*, a hybrid analysis system for soundly enforcing security and reliability policies for JavaScript programs [29]. It employs code instrumentation to detect the presence of runtime code introduction in the program under analysis and then rewrites it to a form that can be handled by a static analysis engine. Gatekeeper cannot be applied to any existing JavaScript program as, even after code rewriting, it is not able to handle all the constructs of the language and focuses instead on providing a sound analysis of a subset of it named $Javascript_{safe}$.

Wei et al. propose *blended taint analysis*, an approach that combines static and dynamic analysis to detect data integrity violations that can make web applications not secure [30]. Blended taint analysis involves human interaction, as a web tester has to manually interact with the application during the dynamic analysis phase in order to collect execution traces. Traces are next leveraged by a static analyzer to build and perform taint-analysis on a control flow graph of the application under analysis. The approach also assumes the presence of a test suite providing program coverage information.

Similarly, Tripp et al. combined black-box testing and static analysis to detect JavaScript security vulnerabilities [31]. Their approach relies on web crawler to collect webpages, complete with code that is dynamically loaded or generated. Their work as been implemented as the JSA analysis tool inside the IBM AppScan suite and evaluated on a collection of 675 real-world websites. Unlike our approach, theirs requires a suite of client-side tests to properly combine the dynamic analysis with the static one.

JSNose is an automated metric-based approach to detect JavaScript code smells in web applications [32]. Dead code elimination is one of the 13 code smells detected by JSNose. To detect dead code inside the application under analysis, JSNose relies on execution traces collected during exploration via an automated crawler. While JSNose combines the results of static and dynamic analysis for the detection of many code smells, it does not for dead code and simply reports every block of code unreachable by the dynamic *or* the static analysis as possible dead code. This leads to a high number of false positives.

VII. Conclusions and Future Work

We presented *Lacuna*, an approach for removing dead code from JavaScript web applications by combining existing program analysis techniques. *Lacuna* is extensible, and any program analysis technique that builds a call graph of JavaScript source code can be straightforwardly integrated in its analysis pipeline.

We evaluated *Lacuna* on a dataset of 29 publicly available web apps, composed of a total of 15,946 JavaScript functions, and developed by professional web developers. The results of the performed evaluation show that the execution time of *Lacuna* is reasonable, making it a good candidate for being integrated in real-world build systems. We also discovered that the combination of multiple analysis techniques via *Lacuna* increases its precision compared to using a single analysis technique in isolation.

As future work, we plan integrating additional analysis techniques and tools to increase the overall effectiveness of *Lacuna*. To this aim, a more in-depth experimentation will be performed to better understand the impact of additional analysis techniques on the overall accuracy of *Lacuna*. ES6 modules are currently not supported by *Lacuna*, mainly because we want to focus on those JavaScript constructs that are widely supported by todays browsers. When ES6 modules will start to be widely supported, we will support them in *Lacuna* and take advantage of the concept of module in JavaScript when combining call graphs. Many unit testing frameworks exist for JavaScript (e.g., QUnit, Mocha, Jasmine), which allow developers to write test suites for verifying their JavaScript code base from a functional perspective. We plan to integrate the information extracted from (unit) tests with *Lacuna* to increase its accuracy in dead code identification.

Finally, we will devise techniques for improving the precision of *Lacuna*, even if we will need to trade off recall. Indeed, having a false negative in *Lacuna* means that we will miss a chance for optimization because a true dead function will not be eliminated. Differently, a false positive will cause *Lacuna* to eliminate an alive function, actually injecting a bug into the analysed web app.

References

[1] Ivano Malavolta. Beyond native apps: Web technologies to the rescue! In *Proceedings of the 1st International Workshop on Mobile Development*, pages 1–2, 2016.

[2] Miguel Ramos, Marco Tulio Valente, Ricardo Terra, and Gustavo Santos. AngularJS in the wild: a survey with 460 developers. *arXiv preprint arXiv:1608.02012*, 2016.

[3] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1663–1671. ACM, 2014.

[4] Ivano Malavolta, Stefano Ruberto, Tommaso Soru, and Valerio Terragni. Hybrid mobile apps in the google play store: An exploratory investigation. In *ACM International Conference on Mobile Software Engineering and Systems*, pages 56–59, 2015.

[5] Ivano Malavolta, Stefano Ruberto, Valerio Terragni, and Tommaso Soru. End users' perception of hybrid mobile apps in the google play store. In *IEEE International Conference on Mobile Services*, pages 25–32, 2015.

[6] N. Palamarchuk J. Vepsalainen, M. Bodnarchuk. A catalog of frontend javascript libraries. http://jster.net.

[7] H. Xi. Dead code elimination through dependent types. In *International Symposium on Practical Aspects of Declarative Languages*, pages 228–242. Springer, 1999.

[8] Hyukwoo Park, Myungsu Cha, and Soo-Mook Moon. Concurrent JavaScript parsing for faster loading of Web apps. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4):41, 2016.

[9] Sacha Greif. The state of javascript developer survey. http://stateofjs.com, 2016.

[10] Ariya Hidayat. Lazy Parsing in JavaScript Engines. https://ariya.io/2012/07/lazy-parsing-in-javascript-engines.

[11] Marja Hölttä and Daniel Vogelheim. New JavaScript techniques for rapid page loads. https://blog.chromium.org/2015/03/new-javascript-techniques-for-rapid.html.

[12] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *ACM Sigplan Notices*, volume 45, pages 1–12, 2010.

[13] Ravi Chugh, Jeffrey A Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. *ACM Sigplan Notices*, 44(6):50–62, 2009.

[14] Simon Holm Jensen, Peter A Jonsson, and Anders Møller. Remedying the eval that men do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 34–44. ACM, 2012.

[15] Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. Eval begone!: semi-automated removal of eval from javascript programs. *ACM SIGPLAN Notices*, 47(10):607–620, 2012.

[16] Oskar Segersvärd Brian Donovan, Bogdan Chadkin. rollup.js. http://rollupjs.org/.

[17] Dr. Axel Rauschmayer. Tree-shaking with webpack 2 and babel 6. http://www.2ality.com/2015/12/webpack-tree-shaking.html.

[18] ES6 modules support. http://caniuse.com/#search=module.

[19] Google. Closure compiler. https://developers.google.com/closure/compiler/.

[20] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[21] Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 121–132. ACM, 2014.

[22] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *International Static Analysis Symposium*, pages 238–255. Springer, 2009.

[23] IBM Research. T.j. watson libraries for analysis. http://wala.sourceforge.net/wiki/index.php/Main_Page.

[24] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. *ACM SIGPLAN Notices*, 46(10):119–138, 2011.

[25] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *European Conference on Object-Oriented Programming*, pages 435–458. Springer, 2012.

[26] Forrest Shull, Janice Singer, and Dag IK Sjøberg. *Guide to advanced empirical software engineering*, volume 93. Springer, 2008.

[27] Daniel Graziotin and Pekka Abrahamsson. Making sense out of a jungle of javascript frameworks. In *International Conference on Product Focused Software Process Improvement*, pages 334–337. Springer, 2013.

[28] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. 1999.

[29] S. Guarnieri and V B. Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium*, volume 10, pages 78–85, 2009.

[30] Shiyi Wei and Barbara G Ryder. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 336–346. ACM, 2013.

[31] Omer Tripp, Pietro Ferrara, and Marco Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 49–59. ACM, 2014.

[32] Amin Milani Fard and Ali Mesbah. JSNOSE: Detecting JavaScript code smells. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 116–125. IEEE, 2013.