Information and Software Technology 107 (2019) 18–29

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Slimming javascript applications: An approach for removing unused functions from javascript libraries

H.C. Vázquez[a,c], A. Bergel[b], S. Vidal[a,c,*], J.A. Díaz Pace[a,c], C. Marcos[a,d]

[a] ISISTAN-UNICEN, Tandil, Argentina
[b] Department of Computer Science (DCC), University of Chile, Chile
[c] CONICET, Argentina
[d] CIC, Argentina

ARTICLE INFO

Keywords:
Javascript
Unused functions
Library dependencies
Software maintenance
Performance overhead

ABSTRACT

Context: A common practice in JavaScript development is to ship and deploy an application as a large file, called *bundle*, which is the result of combining the application code along with the code of all the libraries the application depends on. Despite the benefits of having a single bundle per application, this approach leads to applications being shipped with significant portions of code that are actually not used, which unnecessarily inflates the JavaScript bundles and could slow down website loading because of the extra unused code. Although some static analysis techniques exist for removing unused code, our investigations suggest that there is still room for improvements.

Objective: The goal of this paper is to address the problem of reducing the size of bundle files in JavaScript applications.

Method: In this context, we define the notion of Unused Foreign Function (UFF) to denote a JavaScript function contained in dependent libraries that is not needed at runtime. Furthermore, we propose an approach based on dynamic analysis that assists developers to identify and remove UFFs from JavaScript bundles.

Results: We report on a case-study performed over 22 JavaScript applications, showing evidence that our approach can produce size reductions of 26% on average (with reductions going up to 66% in some applications).

Conclusion: It is concluded that removing unused foreign functions from JavaScript bundles helps reduce their size, and thus, it can boost the results of existing static analysis techniques.

1. Introduction

A JavaScript (JS) application is commonly deployed by *bundling* the source code application with the source of the used libraries. For example, when a website embeds a chart made with Chart.js,[1] the website includes a large `Chart.bundle.js` file, made of concatenating the source code of Chart.js and that of all its dependent libraries. A possible reason for this bundling practice is that the original definition of JS does not consider the notion of module, as other programming languages do.

Although this way of packaging applications is convenient (*e.g.,* only one self-contained file is necessary, no explicit module mechanisms are required), concatenating the application code along with all its dependent libraries tends to deploy more code than necessary. This problem is well-known in the JS community. In fact, tools such as Browserify[2] and Webpack[3] rely on static analysis techniques that can exclude unreferenced and isolated JS modules from a bundle. However, despite the help of such tools, we found evidence that applications are still shipped with unused code related to JS libraries, and our position is that static analysis is not enough for developers to address the problem and thus, dynamic analysis should be also used.

Running a hybrid technique, based on static and dynamic analysis, over a set of JS applications, we detected that bundle size can be reduced by 26% on average (when comparing to the bundles optimized using only static analysis). In this context, we propose the notion of *Unused Foreign Function* (*UFF*) to characterize functions contained in a JS library

[2] http://browserify.org/
[3] https://webpack.github.io/

* Corresponding author.
E-mail addresses: hvazquez@exa.unicen.edu.ar (H.C. Vázquez), abergel@dcc.uchile.cl (A. Bergel), svidal@exa.unicen.edu.ar (S. Vidal), adiaz@exa.unicen.edu.ar (J.A. Díaz Pace), cmarcos@exa.unicen.edu.ar (C. Marcos).

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

https://doi.org/10.1016/j.infsof.2018.10.009
Received 13 December 2017; Received in revised form 18 October 2018; Accepted 22 October 2018
Available online 25 October 2018

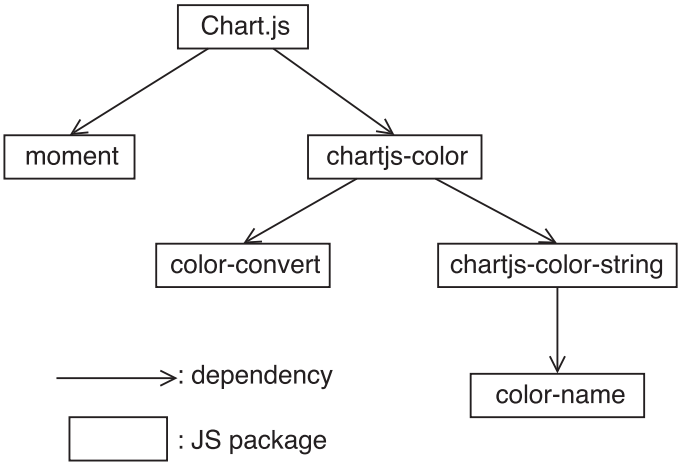

Fig. 1. Chains of dependencies in Chart.js.

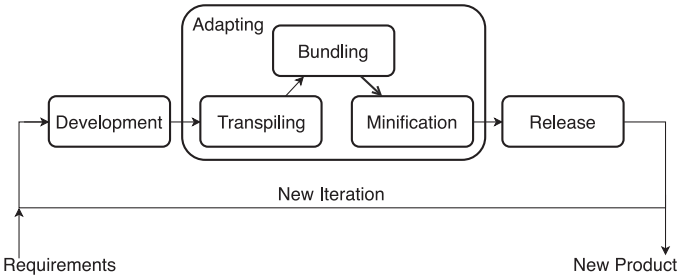

Fig. 2. Development process overview of a JS package.

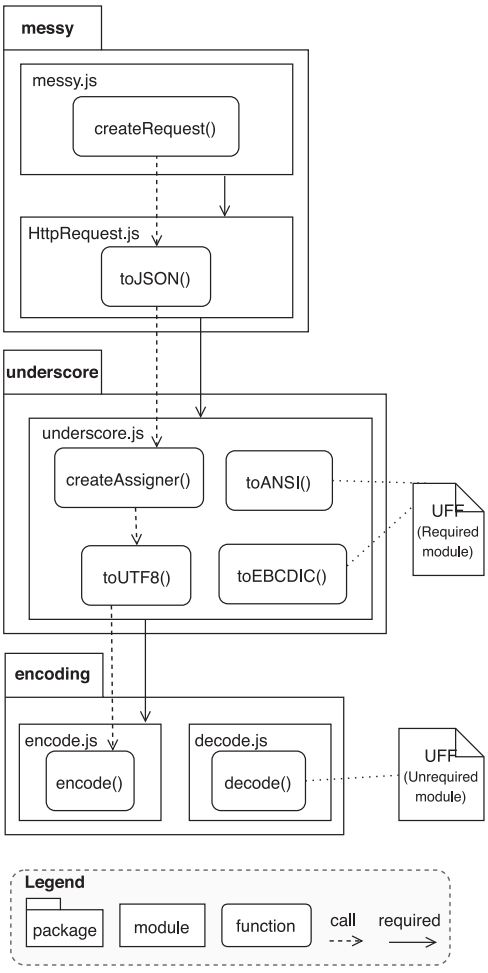

Fig. 3. *UFF* context example.

used by a given application, which are not called by the application, but are still shipped as part of the application bundle. That is to say, *UFFs* are unused functions that do not belong to the application code but to the source code of the dependencies of the application. The fact that these functions are in the source code of dependencies made *UFFs* difficult to be identified.

We have developed a minimizing tool, called UFF Remover, that complements traditional bundling tools by identifying and removing *UFFs* through static and dynamic analysis. Our approach works in 4 stages, as follows. First, we compute the list of all required clusters of source code (called modules in JS). Second, these modules are instrumented. Third, a set of execution traces are obtained from the application so as to identify parts of the libraries that are not used (*i.e.,UFFs* contained in libraries). Fourth, the *UFFs* just spotted are suggested to the developer. Thus, developers can decide which *UFFs* should be automatically removed by our approach. Note that an application may use more than one module system to express dependencies (*e.g.,* npm, umd, CommonJS), and thus, removing unused portions of a library is a challenging activity that, if not carefully performed, can impact the application semantics. We ran our tool over 22 JavaScript applications and found that around 70% of the functions in the bundles that take part in dependencies are *UFFs*. Furthermore, we performed a qualitative study with 10 JS developers that confirmed the usefulness of our approach.

The rest of the paper is structured as follows. Section 2 presents the main concepts used in this work. Section 3 outlines the problem we are focusing on and presents the core steps of our approach. Section 4 describes the case study we conducted and highlights the benefits of our minimizing tool. Section 5 presents the related work. Section 6 concludes and outlines our future work.

2. Background

Functions are a fundamental modular unit in JavaScript (JS) applications [1]. They are generally contained in `.js` files. A function encloses a set of statements and can be invoked from other functions. As with other programming languages, functions are basic bricks that enable code reuse, information hiding, and composition.

Functions can be enclosed by modules. Since JS does not provide built-in module mechanisms, the JS users community has built its own module systems to help developers build small units of independent, reusable code at a higher abstraction level than functions (*e.g.,*using global variables, or implementing CommonJS,[4] AMD,[5] UMD,[6] among others). In practice, most JavaScript modules are implemented in a separate JSfile and export functions that can be used by other modules, while maintaining the remaining functions private to the module.

Module definitions are necessary to implement many JSdesign patterns and they are very useful when building non-trivial JavaScript-based applications [2]. These module definitions are supported by some libraries in EcmaScript 5 (ES5), the standard JSis based on. EcmaScript 6 (ES6 or ES2015) provides a built-in module definition (called Harmony)

[4] http://www.commonjs.org.
[5] https://github.com/amdjs/amdjs-api/wiki/AMD.
[6] https://github.com/umdjs/umd.

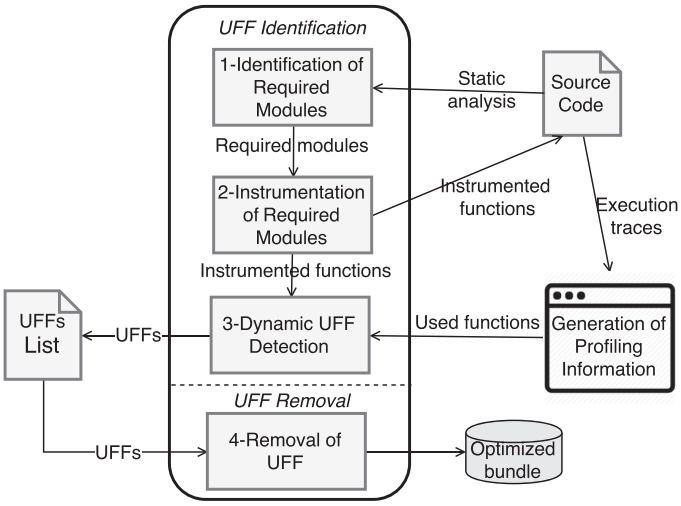

Fig. 4. *UFF* treatment overview.

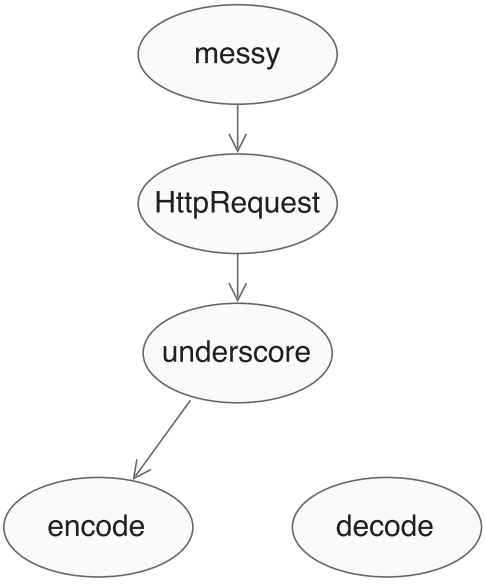

Fig. 5. Module dependency graph example.

but it is not yet completely supported by all JSinterpreters. However, in our work we consider applications written in Harmony because they can be transpiled[7] to ES5.

Moreover, by using any module definition (with the exception of global variables), a given module can specify dependencies on other modules. This kind of dependency is specified by the *required* relationship.

A library is commonly distributed as a ready-to-use single JSfile. However, with the emergence of JS package management systems, libraries are also available as packages. Common JS packages (e.g. Chart.js, Moment, Angular, etc.) are available through central repositories and are handled by package managers such as NPM[8] and Bower.[9] They are similar to other package managers such as Maven[10] in Java or Rubygems[11] in Ruby.

An illustrative example of the use of packages can be seen in the *Chart.js* library. *Chart.js* is a project for creating charts using the HTML5 canvas element. As shown in Fig. 1, functionality about displaying dates and color manipulation of *Chart.js* is delegated to functions implemented in libraries *moment* and *chartjs-color*. Thus, *Chart.js* requires *moment* and *chartjs-color*. In the same way, *chartjs-color* delegates functionality to other library functions. As a result, several chains of dependencies might arise among the libraries.

2.1. Distributing JS applications

Despite the advantages of using external libraries, the use of library functions requires their containing libraries to be included in the deployment environment. In this schema, there are two strategies to perform the deployment. The first strategy is to let the final user download and install the libraries needed. This strategy is generally undesirable because it makes the installation process slower and error-prone. For example, in Fig. 1, the user that needs *Chart.js* should download and install *moment* and *chartjs-color*. Moreover, users may inadvertently install an incompatible version of *moment* or *chartjs-color*, which may cause unexpected failures. Besides, in front-end applications downloading the libraries separately increases the number of HTTP requests, which affects the loading time of the web page [3].

The second strategy is to bundle the required libraries along with the application that uses them. In the example of Fig. 1, this would mean packaging the *moment* and *chartjs-color* libraries with the *Chart.js* code. While this second approach is less likely to incur in version errors or degrade HTTP requests, the size of the distribution can significantly increase. In our example, this means that the size of *Chart.js* with the source code of all its dependencies is significantly bigger than only the source code of *Chart.js* (without libraries). Thus, a single download of a large file is made rather than many small downloads.

2.2. Size of JavaScript bundles does matter

Over the years, front-end JSdevelopers have been choosing the second strategy [4]. One reason for this is that a single HTTP request of a large file is likely to be less expensive than many HTTP requests of small files. In this context, in a typical JS development process, third-party libraries are packaged with the source code being developed before a release. This packaging phase is composed mainly by bundling and minification processes (Fig. 2). These processes are important to improve the deployment and reduce the final size of the application. Reducing the size of JS applications is important because it decreases download times and the amount of data for applications distributed via the internet [5]. Furthermore, it decreases the memory required and power consumption in mobile devices [6].

The bundling process is the concatenation of all JS files into a single file called “bundle”. The bundle contains all the application code and the libraries required by the application. Some bundling tools (such as browserify and webpack) take advantage of this process to reduce code. For example, these tools can discard modules that are not required by any other module (by looking at the *required* relationship) or avoid a package that is not needed and was erroneously included as a dependency of the project.

The bundling process is followed by the minification process. Minification is primarily based on text techniques (e.g. changing names, removing spaces, line breaks and comments) to reduce the final size of the bundle. Some tools (such as Google Closure Compiler[12] (GCC)) offer aggressive compression code transformations, renaming of symbols

[7] Transpilation is the process of transforming and compiling from one language to another with a similar level of abstraction. For example, in JS this task can be performed with Babel.

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

[9] https://bower.io/.

[10] http://maven.org.

[11] https://rubygems.org/.

[12] https://developers.google.com/closure/compiler/.

and dead code elimination through static analysis. However, these approaches do not detect *UFFs*.

Despite the reductions that can be made during the bundling process, we have found that there is still a significant portion of unused code that is shipped when deploying JavaScript applications. Bundling an application with unused code may be problematic if a bundle is large and contains a large amount of code that is not necessary in the context of the application being built. Removing unused code is challenging, partially due to the dynamically typed nature of JS. Determining function calls often depends on the execution of the application along with the context in which it is executed [2,7]. We argue that through a dynamic analysis of the application it is possible to find significant portions of unused code that are not being eliminated during the development process. In this context, monitoring JS executions to identify unused code can greatly complement current bundling tools to reduce the size of the bundles.

3. Reducing bundled application size

We present an approach, called UFF Remover, that complements static analyzes approaches (used during the bundling process). It dynamically analyzes the execution to identify and remove unnecessary source code from libraries, thus, reducing the size of bundle distributions. Specifically, our work focuses on removing *unused functions* from libraries.

3.1. Motivating example

Let us consider the example shown in Fig. 3, which is a simplified situation found in one of our case studies. An application called *messy* (an object model for HTTP messages) uses a third-party library (i.e. a package) called *underscore*. In turn, *underscore* uses a library called *encoding*. The *messy* package contains a main module *messy.js*, and another module called *HttpRequest.js* with functions *createRequest* and *toJSON*, respectively. The *messy* module requires the *HttpRequest.js* module and when the *createRequest* function is called, it subsequently calls the *toJSON* function. Once *toJSON* is invoked, it triggers a series of calls involving the library functions *createAssigner, toUTF8*, and *encode*, by following the chain of dependencies. In this way, all library functions are called except *toANSI* and *toEBCDIC* (from underscore) and *decode* (from encoding). In the context of this example, we argue that functions *toANSI, toEBCDIC*, and *decode* are unnecessary for the application *messy* because they are never executed (at runtime). An optimal bundling process should discard those functions. Existing bundling tools can remove functions from the unrequired modules (such as *decode.js*), but they do not take any action with unused functions from required modules (such as *toANSI* and *toEBCDIC*). For this reason, this work focuses on detecting and removing such functions from required modules.

```
function toJSON(value) {
  return JSON.stringify(value);
}
```

3.2. Approach in a nutshell

Removing *UFFs* contributes reducing the final size of the distribution files without affecting the application behavior. An overview of our approach is shown in Fig. 4. The approach has two stages: (i) the *UFF*identification and (ii) the *UFF*removal. The *UFF*identification stage consists of determining whether a library function is unnecessary in the context of an application. The *UFF*removal stage restructures the source code to remove the *UFFs*. These stages are detailed in the next sub-sections.

3.3. UFF identification

This stage is divided into three main activities, namely: *Identification of Required Modules, Instrumentation of Required Modules*, and *Dynamic UFF Detection*.

1 - Identification of required modules . In this activity a static analysis of the source code of a JSapplication is performed to identify the required modules. Coming back to the example of Fig. 3, all the modules are required by *messy* with the exception of *decode*. To identify required modules, we rely on the Browserify bundling tool. Basically, Browserify traverses the dependencies between modules discarding those that are not required. Thus, all discarded modules will not be part of the final bundle. For example, Fig. 5 shows the dependency graph for the scenario depicted in Fig. 3 in which *messy.js* is the source node. Note that *decode* is not part of the graph (i.e., it has no edges) since it is not required by any module in the source code. All the connected nodes in the graph are required modules and constitute the input for the next activity.

2 - Instrumentation of required modules . This activity instruments all the functions within the required modules (as detected in the previous activity) with the goal of collecting information about those functions effectively executed at runtime. The process of instrumentation starts by parsing the JSfiles in order to identify functions. According to the standard EcmaScript 5[13] (the standard that most of the browsers support), we analyze two types of function patterns: *Function Declaration* and *Function Expression*.

A Function Declaration defines a named function variable without requiring variable assignment. Function Declarations occur as standalone constructs and cannot be nested within non-function blocks. The syntax is defined as **function** *Identifier* (*FormalParameterList*$_{opt}$) {*FunctionBody*}. For example:

A Function Expression defines a function as a part of a larger expression syntax (typically a variable assignment). Functions defined via Function Expressions can have a name or be anonymous (i.e. the only difference with a Function Declaration is that the identifier is optional). The syntax is defined as **function** *Identifier*$_{opt}$ (*FormalParameterList*$_{opt}$) {*FunctionBody*}. For example:

[13] http://www.ecma-international.org/ecma-262/5.1/.

```
//anonymous function expression
var toJSON = function(value) {
  return JSON.stringify(value);
}

//named function expression
var jsonfy = function toJSON(value) {
  return JSON.stringify(value);
}

//self invoking function expression
var value = "msg:hello";
(function toJSON() {
  alert(JSON.stringify(value));
})();
```

Once the functions are identified, the instrumentation adds an instruction at the beginning of each function defined in a library. This instruction logs information into a file at runtime, recording whether a function was executed. In this way, the output of the activity are all the instrumented functions of the required modules of libraries. It is important to remark that this instrumentation is automatically made by our approach.

To generate the information about executed functions, the program needs to be exercised in the environment for which it was developed through execution traces. This is achieved through the task *Generation of Profiling Information*. The execution traces must ensure a program coverage as complete as possible. This task can be performed using the tests shipped with the program (if any) in the development environment or via interactions with the program in the production environment. Considering the example of Fig. 3, a function call *toJSON()* would produce the following execution trace:

```
function createAssigner() defined in /underscore/underscore.js was executed
function toUTF8() defined in /underscore/underscore.js was executed
function encode() defined in /encoding/encoding.js was executed
```

Note that functions *createAssigner()*, *toUTF8()*, and *encode()* were executed as a result of the invocation to function *toJSON()* in module *HttpRequest.js*.

The output of the *Generation of Profiling Information* task is a trace file that contains all the functions of required libraries called during the execution of the application.

3 - Dynamic UFF detection .This activity analyzes the information collected at runtime about function executions. For each function in the required modules of the libraries, this activity checks whether a trace exists that indicates that the function was executed. If the function is not found in the trace file (i.e. it was not executed), it is classified as *UFF*. In the context of the example of Fig. 3, functions *toANSI* and *toEBCDIC* are never called, thus a trace indicating their execution is never logged in the file. Therefore, *toANSI* and *toEBCDIC* are identified as *UFF*. At last, a list with all the *UFFs* is returned to the developer.

3.4. UFF Removal

Once a *UFF*is confirmed by the developer, our approach helps to remove it. Eliminating *UFFs* is not always straightforward in real-life applications. This is due to the extensive use that JSdevelopers give to dynamic features of the JSlanguage [8]. An example of a dynamic feature is the *eval* function. Eval is widely used by developers and it has the capability of executing code provided as a string, making it a powerful mechanism of reflection. Let us assume that function *createAssigner* in module *underscore.js* uses *eval* to create an object:

```
//underscore.js module
function toUTF8(value){
  //... some code to encode as UTF−8
}
function toANSI(value){
  //... some code to encode as ANSI
}
function toEBCDIC(value){
  //... some code to encode as EBCDIC
}
function createAssigner(assignTypes){
  var objToCreate = "var assigner = {";
  for (i in assignTypes) {
    objToCreate = objToCreate + " " + "assign"+assignTypes[i] + ":" + "to"+
    assignTypes[i];
    if(i==(assignTypes.length−1)){
      objToCreate = objToCreate + " } ";
    }else{
      objToCreate = objToCreate + " , ";
    }
  }
  eval(objToCreate);
  return assigner;
}
//...

//HttpRequest.js module
var assigner = createAssigner(getAssignerTypes());
//getAssignerTypes load from a property file an array with the values ["UTF8","ANSI"]
//...
//In messy.js module
assigner.assignUTF8(valueToEncode);//call to toUTF8 function
//...
```

In the code above, our approach identifies functions *toANSI* and *toEBCDIC* as *UFFs* since they are not executed by the application. However, if we remove these functions from the source code, it will crash in the next execution. This is because the *eval* function is used to construct an object whose methods are determined at runtime. The method *createAssigner* creates an object with properties *assign+assignTypes[i]* binding to methods *to+assignTypes[i]*, where assignTypes[i] is part of the name of a function. The names are passed as parameter ("UTF8","ANSI") from the module *HttpRequest.js* that loads them from a properties file. Thus, while the application only executes the function to encode in UTF8, if the function "toANSI" does not exist, the execution throws an exception when *eval* is executed. This is because the interpreter will try to bind a property *assignANSI* to a method *toANSI*. Although the *UFFtoEBCDIC* could be eliminated from the final bundle, it is very difficult to distinguish the *UFFs* that are referenced using dynamic features of JSfrom those that are not.

For this reason, our approach uses a less aggressive strategy. Specifically, our approach automatically "empties" the functions instead of removing them completely. This strategy preserves the application behavior in situations like the one above. Also, since the source code generated by our approach is not intended to be read by a developer but to be minified (as in Uglify), this strategy does not affect the readability of code. However, emptying a function has a disadvantage. The code related to the header and keys for opening and closing the body of the function are preserved taking up space. Moreover, to make our removal strategy safe, we use a lazy load mechanism for the removed functions. This mechanism is implemented by replacing the body of an *UFFs* by a synchronous XMLHttpRequest[14] when emptying the function. If the function is called at run-time (i.e. the *UFF* is a false positive), this call loads the removed body from the server. While this strategy makes the removal safe, it has the disadvantage of adding a line of code with the call and also a global function with the code related to the lazy load. Nevertheless, we believe that this space is negligible regarding the total size of the function after the minification process. Another disadvantage of this strategy is that it can affect the loading time of the application, in case of having a large number of false positives. However, these false positives could be removed by the developer in subsequent bundles by indicating to UFFRemover to not remove these *UFFs*. An example of the resulting code is shown below, in which functions *toANSI* and *toEBCDIC* are emptied:

[14] This kind of request is usually enabled by AJAX technology. It is important to remark that some browsers can discontinue this feature due to UX problems. However, this lazy loading could be replaced with a similar technology in the future.

```
//In underscore.js module
//...
function toUTF8(value){
//... some code for encode as UTF−8
}
function toANSI(value){
  eval(_loadUFFScript(toANSI.uff))
}
function toEBCDIC(value){
  eval(_loadUFFScript(toEBCDIC.uff))
}

// Global Function
function _loadUFFScript (uffScript){
   var xhReq = new XMLHttpRequest();
   xhReq.open(\"GET\", uffScript, false);
   xhReq.send(null);
   return xhReq.responseText;
}
```

A special case of *UFFs* found dynamically are those functions being declared inside the scope of an *UFF*. Nested functions declared in an *UFF*are also *UFFs*. Nested functions can be completely removed, but the *UFF*that contains the nested functions should be emptied.

The output of this activity is the source code of all required modules optimized by removing the bodies of detected *UFFs*. As described later on, removing UFFs by simply emptying them is practical, easy to implement, and cope with all the reflection mechanisms used in our large set of studied applications.

4. Evaluation

In this section, we present an empirical analysis about *UFFs* in JS-applications. First, we describe the research questions (4.1). Second, we introduce the target applications (4.2). Third, we analyze the number of *UFFs* in the applications (4.3). Fourth, we present the results of applying our removal strategy on the *UFFs*(4.4). Fifth, we present a qualitative study with 10 JSdevelopers that employed our approach in their own applications (4.6). Finally, we discuss the threats to validity of our study (4.7).

4.1. Research questions

In order to understand the phenomenon of *UFFs* and to evaluate our approach, we formulate two research questions:

- RQ1: What is the number of *UFFs* in JS applications?
- RQ2: How much can an application source code be reduced, if its *UFFs* are removed with our approach?

The goal behind these questions is to determine the applicability of our approach.

4.2. Target applications

We selected 22 JS applications. Each application must meet the following conditions:

- It must be open-source.
- It is runnable in a Web browser.
- Test coverage of the functions of the application must be greater than 85%. This is because, in order to generate execution traces, we will rely on the unit tests shipped with the applications. We think that using tests for this task instead of interacting with the applications in the production environment, will allow others to reproduce our experiments. Along this line, it is important to count with a high coverage of tests to ensure an application coverage as complete as possible.
- It must depend on at least one library.
- It must have information about how to compile it and run its tests.

Table 1 lists the set of applications and their main characteristics. The table contains all the software versions to let the interested reader reproduce our findings. Additionally, the implementation of our approach is available for download.[15]

Table 1 shows not only the direct libraries on which each application depends but also the indirect dependencies being generated by the direct dependencies (all the libraries resulting from these dependencies are part of the bundle). For instance, while *geojsonhint* only depends directly on 5 libraries, *geojsonhint* indirectly depends on 19 libraries. Thus, *geojsonhint* depends on a total of 24 libraries.

Another interesting fact from Table 1 is the percentage of Lines of Code (LOC) from the bundle (no blank or comment lines) that belongs to dependent libraries. While the total number of LOC of all bundles is 336,983, for which 158,695 belong to dependent libraries (47.09%) . Something similar happens with the number of functions. A total of 19,586 functions are defined in the bundles, but 10,063 of them belong to libraries (51.38%). Thus, since the incidence of libraries in the bundles is large, it is important to analyze the functions that belong to dependent libraries for removing unnecessary functions. Finally, Table 1 also reports for each application the size of the minified bundle. In order to create a baseline to compare the different applications, we used Browserify to bundle the JSfiles and Uglify[16] to minify the bundles. That is, the bundle sizes reported on Table 1 are already minified. Thus, our results (cf Section 4.3) show the improvement regarding the existing static analysis techniques (i.e. minifiers).

4.3. UFF identification

In order to identify the *UFFs* of the applications using our approach, we followed 7 steps for each application:

[15] https://github.com/hcvazquez/UFFRemover.
[16] http://lisperator.net/uglifyjs/.

Table 1
Applications used in our study.

Application	Version	Test coverage	Direct dependencies	Indirect dependencies	Bundle LOC	Dependencies LOC in bundle	#Functions in bundle	#Functions only in dependencies in bundle	Minified bundled size (bytes)
angular-countdown	1.2.1	100%	2	0	24,094	23,600	1219	1168	136,697
assert-x	1.2.18	89%	6	56	5652	5239	328	306	46,411
backbone-tableview	1.0.5	100%	4	1	11,350	7770	978	916	136,054
Chart.js	2.1.4	91%	2	3	13,728	6394	1044	447	187,074
chomsky	1.0.8	90%	4	3	20,029	15,530	2064	1643	242,023
easystarjs	0.3.0	100%	1	0	916	318	72	31	8,769
escodegen	1.8.1	97%	5	7	4946	3714	370	196	99,161
escope	3.6.0	90%	4	8	7565	3125	604	220	97,853
geojsonhint	2.0.0	97%	5	19	2022	1097	109	33	30,801
mathjs	3.5.1	96%	5	0	50,885	6841	2865	281	507,702
messy	6.11.0	92%	5	2	22,405	8653	1069	504	441,294
mochawesome	1.5.2	100%	12	15	11,255	11,077	936	920	154,656
p5	0.5.3	100%	3	44	34,985	8585	1132	307	281,130
pixi.js	4.0.1	86%	8	73	26,527	4,043	1402	211	300,256
teoria	2.2.1	86%	5	2	1766	409	118	11	26,688
transform-pouch	1.3.3	90%	8	48	1467	970	214	134	27,899
underscore.string	3.3.4	100%	2	0	1600	245	170	11	31,935
unexpected-http	5.6.0	87%	3	13	39,037	25,857	1923	1516	584,046
unexpected-messy	6.1.2	92%	4	10	36,942	21,725	1766	1114	557,454
unified	5.1.0	100%	8	4	2040	722	161	54	29,639
virtual-dom	2.1.1	100%	8	6	1565	150	112	8	31,181
workfront-api	1.3.4	89%	3	6	16,207	2631	930	311	226,744
	Total				336,983	158,695	19,586	10,063	4,185,074

1. Download the source code of the application (this step usually requires to download all the dependencies).
2. Create the bundle from the source code using Browserify and Uglify.
3. Run all the tests and verify that they all pass.
4. Run the *Identification of Required Modules* of our approach.
5. Run the *Instrumentation of Required Modules* of our approach.
6. Run all the tests to log the traces using the instrumented code.
7. Run the *Dynamic UFF Detection* activity.

When instrumenting the source code and running the *Dynamic UFF Detection* activity, we found a total of 7021 *UFFs* in the bundles (Table 2). That is, 69.77% of the functions in the bundle that belong to dependent libraries are *UFFs* (35.85% of the total number of functions in the bundle) . Moreover, these *UFFs* are responsible for a high number of lines of code of the bundle. Specifically, *UFFs* represent 54.35% (=86,249) of the LOC in the bundle that belongs to dependent modules (25.59% of the total number of LOCs in the bundle).

To answer RQ1, we plotted the percentages of the number of *UFFs* with respect to the total number of functions of each bundle (Fig. 6). The median percentage is 27.56% and the inter-quartile range is 14.69%-47.17% (i.e. 50% of the applications analyzed are in this range). Some of the applications are outliers such as *teoria* and *virtual-dom* that only report a 1.7% of *UFFs*. After manually analyzing their source code we found that the low percentage of *UFFs* is due to the fact that both applications depend on very few functions (*teoria* has in its bundle only 11 functions that belong to dependencies and *virtual-dom* has only 8 functions). Moreover, we tested for a correlation between the percentage of functions in the bundle belonging to dependencies and the percentage of *UFFs*. In order to run a statistical test, we tested the data for normality using the Shapiro–Wilks test and concluded that the data deviates from normality (p-value = 0.015). Thus, we used the Spearman's correlation, and we obtained a value of 0.92 (p-value = 4.005e-6). Thus, the percentage of *UFFs* is directly correlated with the percentage of functions of depending libraries. The greater the percentage of such functions in the bundle, the greater the percentages of *UFFs*. This result seems to imply that the use of libraries in JS applications increments the unused code, justifying the need of removing *UFFs*. As we mention earlier (see Section 2.2), unused code can increase the downloading time of a website and can negatively impact on user experience.

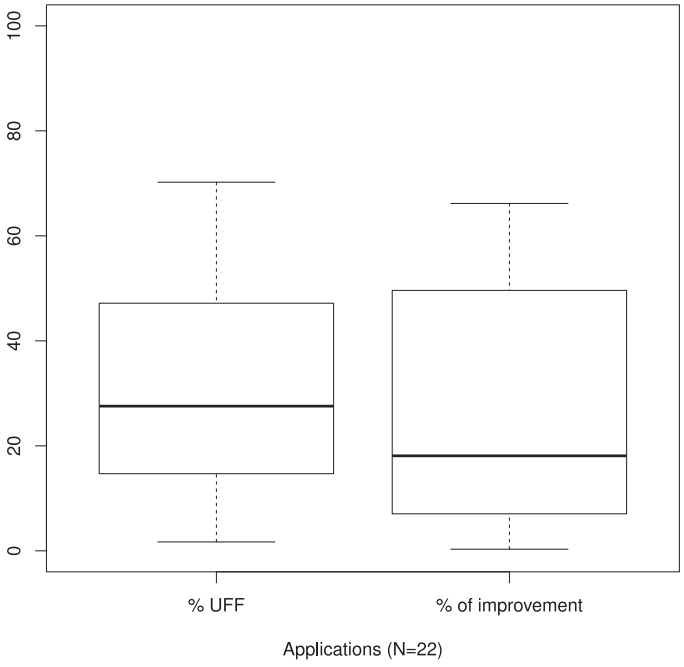

Fig. 6. Results *UFFs* identification and removal.

4.4. UFF removal

To evaluate the removal activity of our approach we followed a series of steps for all the *UFFs* previously identified:

Table 2
UFFs identified and removed.

Application	#UFFs detected dynamically	LOC of UFFs detected dynamically	#UFFs removed	#UFFs emptied	LOC of UFFs removed/ emptied	Minified improved bundle size (bytes)	% of reduction
angular-countdown	856	7237	471	385	6852	57,680	57.80%
assert-x	90	528	17	73	455	37,929	18.28%
backbone-tableview	632	5307	163	469	4838	68,551	49.61%
Chart.js	289	1843	4	285	1558	160,916	13.98%
chomsky	1,054	6532	94	960	5572	149,236	38.34%
easystarjs	19	155	1	18	137	7358	16.09%
escodegen	107	1139	25	82	1057	81,392	17.92%
escope	201	2219	73	128	2091	70,352	28.10%
geojsonhint	14	102	0	14	88	28,628	7.05%
mathjs	72	702	1	71	631	500,731	1.37%
messy	361	12,696	174	187	12,509	149,289	66.17%
mochawesome	631	5537	203	428	5109	73,690	52.35%
p5	301	4965	210	91	4874	207,024	26.36%
pixi.js	206	2298	26	180	2118	269,026	10.40%
teoria	2	4	0	2	2	26,605	0.31%
transform-pouch	78	363	41	37	326	20,454	26.69%
underscore.string	6	40	2	4	36	31,277	2.06%
unexpected-http	993	16,395	211	782	15,613	232,836	60.13%
unexpected-messy	833	16,621	255	578	16,043	215,361	61.37%
unified	25	129	5	20	109	26,826	9.49%
virtual-dom	2	8	0	2	6	31,058	0.39%
workfront-api	249	1429	166	83	1346	211,619	6.67%

1. Run the *Removal of UFF* of our approach to optimize the application.
2. Create the bundle from the optimized source code using the instructions and tools provided by the application.
3. Run all the tests and verify that they all pass.
4. Test the optimized application with a third-party application that uses the analyzed application. To accomplish this step, we manually replace (in the third-party application) the original version of the analyzed application for its reduced version. Then, the tests of the third-party application are run.

Table 2 shows the results of our experiment. The 7021 *UFFs* were removed or emptied by our approach. Specifically, 30.5% of the *UFFs* (= 2142) were completely removed while the bodies of 69.5% of them (=4879) were emptied. This procedure allows us to remove a total of 81,370 lines of code from the bundles. This corresponds to 94.34% of the source code identified as *UFF*. Moreover, after analyzing the impact of the *UFF*removal in the minified version of the bundles, we found that the inter-quartile range of reduction is 7.66%–46.80% with a median of 17.99% (Fig. 6).

Some applications are above this range, such as the case of *Messy* that was reduced by 66.17% while its percentage of *UFFs* was only 33.77%. After carefully analyzing the bundle of *Messy* we found that this high reduction was accidental since the *UFFs* removed were, in general, long functions.

Also, to understand the relationship between the percentage of *UFFs* and the reduction in the minified bundle after removing them. In order to run a statistical test, we tested the data for normality using the Shapiro–Wilks test and concluded that the data deviates from normality (p-value = 0.005). Thus, we used the Spearman's correlation, and we obtained a value of 0.91 (p-value = 3.734e-6). We found that there exists a strong correlation of 0.91 between these factors. Thus, the greater the percentages of *UFFs*, the larger the bundle reductions.

Since the size of the bundles are reduced, these reductions should improve the loading time of web-pages using them. While an empirical experimentation of loading times is out of the scope of this paper, we conducted a small sanity check to test the download time. First, we created a simple web-page[17] that uses the *Math.js* application. Specifically, the web-page prints a number of mathematical operations using a subset of the functions implemented in *Math.js*. Also, we used an example of a bar chart taken from the website of *Chart.js*.[18] Second, we applied our approach to the minified *Math.js* and *Chart.js* files to obtain the optimized versions of them. We obtained the execution traces by simply executing the web-pages. In the case of *Math.js*, we obtained a reduction of 36.67% (499Kb vs 316Kb). In the case of *Chart.js*, the reduction was around 10.1% (209Kb vs 188Kb). Third, we compared the download time of the web-pages using the original and the optimized bundles. We hosted the web pages locally. To reduce the bias introduced by the latency of the http server, we loaded 10 times the web-pages with each bundle. In the case of *Math.js*, we found an average reduction of 9% of the total download time (13.64% when only the bundle downloading time is considered) while in the case of *Chart.js*, the reduction was of 8.5%. Thus, both web-pages experienced reductions in the download time. However, given the complexity of these kind of analysis a deeper empirical analysis should be conducted in future works to analyze the loading time of websites. These analysis are complex because a number of variables can affect the loading time: download time, parse time, network bandwidth, JSinterpreter, among others [9].

In order to assure that our removal strategy does not affect the behavior of the applications, we followed a two-step validation process. First, we ran again the tests of the application to check that they still passed. Second, in third-party applications we replaced the bundles provided by the application authors with our trimmed and optimized bundle. Our goal with the first validation was to conduct a sanity check to determine if the removal of *UFFs* does not produce compilation or execution errors. The goal of the second validation was to check if all tests passed after optimizing the bundles. We conducted this process in the 22 applications and all the tests passed. Moreover, we checked the number of times that our lazy load mechanism was executed and no call was found.

[17] The source code can be found at https://github.com/hcvazquez/ExperimentExample .

[18] The source code can be found at https://github.com/hcvazquez/BarChartExample .

Table 3
Validation with third-party applications.

Application	3rd application	% of coverage
Messy	unexpected-messy	93
	unexpected-http	100
	unexpected-mitm	90
Mathjs	loess	91
	dn2a	74
	dext	92
	pullquoter	85
Transform-pouch	crypto-pouch	100
	pouch-box	81
Escope	derequire	86
	eslint	100
	fancyscript	96
	isogrammify	100
Escodegen	browjadify	100
	espower	100
	esreflect	100

If the optimized application is intended to be used by third-party applications, those third-party applications could use some functions defined in a dependency of the optimized application. Along this line, the goal of our second validation is to analyze if third-party applications do not change their behavior after replacing the original application by the application optimized by our approach. This validation is challenging because for each optimized application we need to find several third-party applications that use the same version of the optimized application. Moreover, the third-party applications must have a high test coverage. For this reason, we conducted this validation with only 5 applications (we could not find other applications that meets both criteria). Table 3 shows the applications and the third-party applications that use them. After replacing the original library code with the optimized one we did not obtain errors on the tests nor were lazy loads detected. However, an exception occurred when executing *unexpected-mitm* to test the reductions in *Messy*. The problem was that *Messy* uses the *underscore* library, thus, when analyzing *Messy* our approach removed *UFFs* from *underscore*. At the same time, *unexpected-mitm* also uses the *underscore* library. Therefore, some *UFFs* in *Messy* are not *UFFs* in *unexpected-mitm* and as a result an error occurred. However, what we want to show is that *Messy* does not use those functions during its execution in the context of a library that depends on it. In this way, we separate the underscore library during the test, using an optimized version for *Messy* and one without optimization for *unexpected-mitm*. Consequently, the tests were successful, showing that *Messy* does not use these functions in the context of *unexpected-mitm*.

After this analysis, we can answer RQ2 by saying that the applications are reduced with a median of 18% after applying our approach.

4.5. Impact of low test coverage

Our approach uses execution traces to identify *UFFs*. The question that naturally arises is how the coverage of the execution traces can affect the effectiveness of the approach. We hypothesize that the percentage of coverage of the execution traces inversely correlates to the number of false positives identified by our approach. That is, the lower the coverage, the higher the number of false positives. However, these false positives should not variate the program behavior because of the dynamic loading strategy of our approach. To answer this question, we analyzed *chartjs-color*, which is a dependency of *Chart.js* (Fig. 1). We selected *Chartjs-color* because it has a high test coverage (93.44%) and also has dependencies (2 direct dependencies and 1 indirect one). With the goal of analyzing the number of false positives, and possible execution problems related to them, we tested our approach in *Chartjs-color* with different test coverages. We applied 4 steps, namely:

1. Follow steps 1 to 7 described in Section 4.3 for *Chartjs-color*.
2. Follow steps 1 to 4 described in Section 4.4 for *Chartjs-color*.
3. Run all the tests of *Chart.js* using the optimized version of *Chartjs-color* and verify the existence of false positives.
4. Randomly reduce the test coverage[19] of *Chartjs-color* and re-start the process (Step 1).

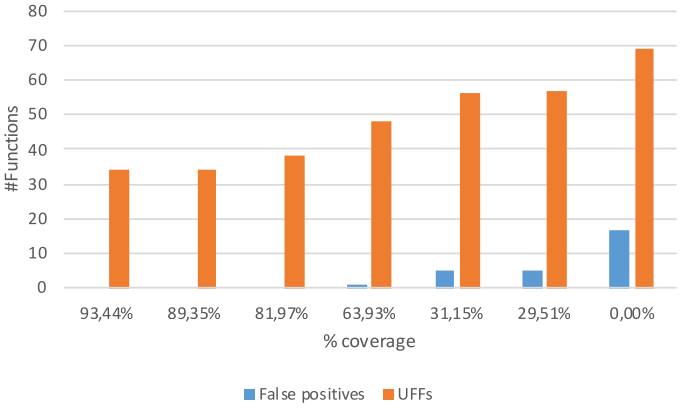

Fig. 7. Relationship between false positives and coverage for *Chartjs-color*.

The results of the experiment are depicted in Fig. 7. Our approach found 34 *UFFs* in *Chartjs-color* after generating the execution traces using the original set of tests (93.44% of coverage). As expected, the number of false positives (detected through the execution of lazy loads) increases as test coverage decreases. While running the experiment, we found the first false positive after reducing the test coverage by 31.5%. We could not find any odd behavior during the execution of the tests. That is, when a false positive *UFF*was invoked, the dynamic loading strategy loaded the removed function.

As it can be seen in Fig. 7, the percentage of coverage of the execution traces is directly related to the effectiveness of the approach. For this reason, developers should be careful when providing traces as inputs for our approach. A possible way to obtain a set of traces is through the analysis of the interaction of users with the application. Since we so far analyzed only one project, the results cannot be generalized to other projects. Similar experiments with other applications are necessary as future work.

4.6. Study with developers

To sense the opinion of JSdevelopers about our approach, we conducted a qualitative study with industrial developers. To this end, we invited industrial developers to complete an on-line questionnaire about UFFRemover, our tool to detect and remove *UFFs*. The invitations were sent via e-mail. The participants received the questionnaire via Google forms,[20] which provided: detailed instructions to do the experiment, a background survey, and a number of tasks to be performed by each participant. A total of 10 developers from different companies participated in the study, most of them (80%) had 5 or more years of programming experience.

The study was composed of two tasks. In the first task, each participant was asked to answer questions about their background in programming and their opinion about identifying and removing unused parts of applications. In the second task, each developer was given a short introduction to the notion of *UFFs* and UFFRemover. Then, developers were asked to use the UFFRemover on their own applications and report their results. We provided developers with a detailed tutorial about our tool.

[19] The reduction is made by randomly selecting and removing a number of unit tests.

[20] The tasks and questions can be found at https://goo.gl/forms/HUedb3sswJQloNOi1 .

At the end of this task, we asked each participant to fill in a post-task questionnaire about *UFFs*and UFFRemover.

Participants were allowed to spend as much time as they need to complete the tasks. Also, we allowed participants to leave optional comments for each task. Next, we analyze the results of the tasks in detail.[21]

4.6.1. Task 1: Developers opinion about unused code

During the first task, developers were asked about their background in programming and their opinion about unused parts of a JSapplication. Regarding the latter, we asked the following questions in which each participant had to answer using a Likert scale ranging from 1 to 5:

1. How important do you think is the identification of parts of an application that are not used? (1=unimportant; 5=very important)
2. How often do you identify unused parts of your application? (1=almost never; 5=very often)

The intention of both questions was to analyze if developers were aware of detecting fragments of unused code. In the case of question #1, 80% of the developers answered that the identification of not used parts is important or very important (2 developers answered 3, 3 developers answered 4, and 5 developers answered 5). Moreover, some developers justified their answers by relating this importance to the performance of the application. For example, some developers said “It’s important to keep a clean code-block, and to make the app as light as possible for the user”, “... it downgrades the performance of the application in the front-end. In the back-end it can affect the maintainability and readability of the code”, “... in web and mobile environments, the size of apps is important for performance”, “Identifying unused code from libraries may reduce network consumption and, thus, produce faster load times”.

However, when answering question #2, most developers answered that they do not often identify unused parts of the applications. Specifically, 8 developers answered between 1 and 3 (1 developer answered 1, 3 developers answered 2, 4 developers answered 3, and 2 developers answered 5). In this context, we think that developers could benefit from our approach.

4.6.2. Task 2: Results of using UFF Remover

In the second task, we asked developers to apply our approach to an application of their choice. After finishing this activity, we asked the following questions in which each participant had to answer using a Likert scale ranging from 1 to 5:

1. How harmful do you think are the *UFFs* for the maintainability of the system? (1=harmless; 5=harmful)
2. How harmful do you think are the *UFFs* for the performance of the system? (1=harmless; 5=harmful)
3. How useful do you consider the tool used in the experiment? (1=unhelpful; 5=very useful)

Fig. 8 shows a bar chart of the answers of the developer for questions #1 and #2. Regarding question #1, 7 of 10 developers answered between 1 and 3. The median of these answers is 3 which means that developers are uncertain about the harmfulness of *UFFs* in the maintainability of the applications.

Regarding question #2, 1 developer answered 1, 8 of 10 developers answered 4 or 5. The median of these answers is 4 meaning that developers estimated that *UFFs* can harm the performance. Thus, developers tend to agree that *UFFs* are not harmful for maintenance but that they are harmful for performance.

Finally, we analyzed how useful the developers consider our approach (question #3). A total of 9 developers answered 4 or 5 (1=unhelpful; 5=very useful). Fig. 9 shows a bar chart with the answers of developers. These results indicate that participants found the approach implemented by UFFRemover useful for removing *UFFs*.

[21] All the results can be found at https://goo.gl/ZHJGF5 .

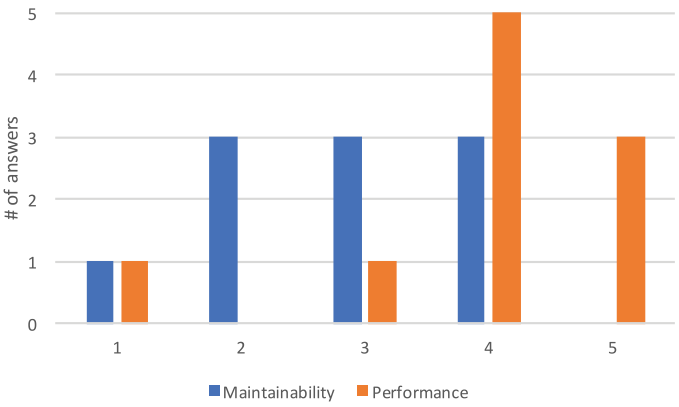

Fig. 8. Harmfulness of *UFFs* for maintainability and performance.

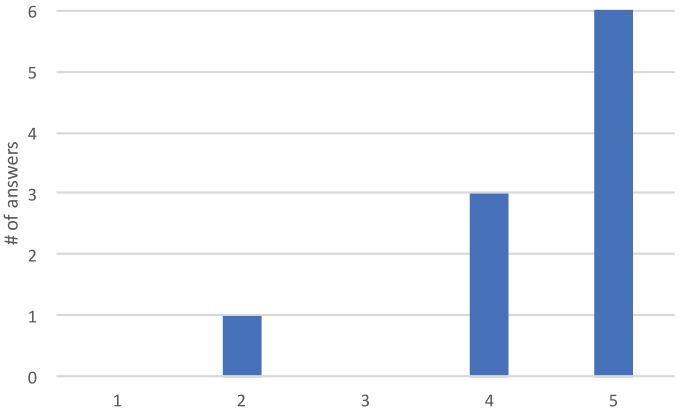

Fig. 9. Usefulness of the approach.

4.7. Threats to validity

The validity of our results depends on factors in the experimental settings. Along this line, we analyze four kinds of validity threats [10].

4.7.1. Conclusion validity

This threat concerns the statistical analysis of the results. The applications analyzed in this work had to satisfy a number of conditions (see Section 4.2) that could not represent the situation of most JS applications. However, we argue that the statistical relevance of the results is appropriate given the number of applications used in the experiment.

4.7.2. Internal validity

This threat concerns causes that can affect the independent variable of the experiment without the researcher’s knowledge. The percentage of tests coverage is an important factor in our study. However, the odds of having false positives with partial test coverage should be considered, because important calls to library functions could go undetected. For example, a programmer might have forgotten to test the code that deals with a particular user-agent or the particular code for a certain platform. We mitigated this threat by selecting applications with a high test coverage (> 85%). Also, it is important to remark that the tests are a mechanism to obtain execution traces, but these traces can alternatively be obtained by other means (e.g. interacting with the application in the production environment).

4.7.3. Construct validity

It is concerned with the design of the experiment and the behavior of the subjects. Our main concern is that the execution traces of an application could not encompass all the possible uses of that application. Thus, some functions could be identified as *UFFs* when they are not. To

mitigate this threat we analyzed the reduced bundle of the applications with third-party applications that use them (Section 4.4). Also, it is possible that some applications were designed only for use in conjunction with other applications that may use some function identified as UFF in the context of the first application. To deal with this threat, we only selected libraries providing a bundle creation process for independent usage.

4.7.4. External validity

It is concerned with having a subject that is not representative of the population. We argue that the number of applications analyzed in the study is large enough to avoid this threat, so the results can be generalized to other JS applications. To mitigate this threat we selected applications with different sizes, purposes and domains.

5. Related work

As far as we are aware of, no empirical study has been conducted on the analysis of unused library functions in JS developments. However, some works have identified relevant issues in the analysis of JS applications.

Since programmers started using JS for writing complex applications, the need for better tool support during development increased [11]. In this context, the main efforts have focused on having a static analysis infrastructure for the full language as defined in the ECMAScript standard. Jensen et al. [12,13] proposed an approach to design a *full-blown* JS program analyzer. This analyzer can be incorporated into an IDE to supply on-the-fly error detection as well as support for auto-completion and documentation hints. There are works [14–16] about understanding and modeling the program dataflow and the interaction between JS and the host environment such as the browser. Other works [7,17] focus on the construction of approximated call graphs between functions. These approaches work well in IDEs, where the precision of the analysis can be relegated to obtain a high response speed. Unfortunately in works that need more precision these studies have difficulties dealing with the dynamic behavior of JS applications. Richards et al. [8] perform an exhaustive study of the dynamic behavior of JS. The study identifies the frequent use of the eval function as one of the main causes that tends to change the semantics of the application at runtime. Fard and Mesbah [18] propose a tool that detect dead code in JSusing dynamic analysis. However, they do not analyze the libraries of an application but the application itself.

In industry, bundling tools as Browserify and Webpack are able to reduce the source code discarding modules that are not explicitly required by the program. However these tool do not remove unused functions from used modules. A recent approach named tree-shaking, implemented by the Rollup[22] tool, can delete function from modules that are not imported through the ECMA6 import/export syntax. However, Rollup does not remove things like unused functions from modules that are not declared with the export syntax, and many times it is forced to assume a function used in order to ensure that the resulting code is correct. A key feature of our approach is that it helps in obtaining information about the function executions at runtime. This information allows us to eliminate unused functions regardless of whether the modules are used, yielding reductions in the final size of the application.

6. Conclusion and future work

Removing unused foreign functions from JS bundles helps reduce their size. We have proposed an approach to detect and remove *UFFs* during the bundling process. We have empirically determined for a set of 22 JS applications that around 70% of the functions that belong to dependencies are *UFFs*. Moreover, we found that around 26% of the bundle sizes can be reduced after applying our approach. We also performed a qualitative study with 10 industrial developers that confirmed the usefulness of the approach.

Although promising, these results are still preliminary and subject to the limitations of our dynamic analysis techniques based on the tests provided by the JS applications. We believe that an ideal approach should integrate both static and dynamic analyses of the bundles as part of the JS development environment. Furthermore, we have looked so far at *UFFs* in libraries that are dependent on the main application. However, if we follow the dependency chain of those libraries, we speculate that *UFFs* can also occur in other libraries. For instance, in Fig. 1, we should not only check for *UFFs* in *color-convert* regarding to *Chart.js* but also check for *UFFs* in *color-convert* regarding to *chartjs-color*.

As future work we plan to:

- Apply our approach in websites to analyze their loading times with bundles in which *UFFs* have been removed.
- Integrate the dynamic analysis traces with static analysis to improve the *UFFs* detection performance.
- Identify further *UFFs* by analyzing the dependency chain of libraries.

References

[1] D. Crockford, Javascript: the good parts: the good parts, O'Reilly Media, Inc., 2008.
[2] M. Madsen, B. Livshits, M. Fanning, Practical static analysis of javascript applications in the presence of frameworks and libraries, in: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering, ACM, 2013, pp. 499–509.
[3] S. Souders, High-performance web sites, Commun. ACM 51 (12) (2008) 36–41.
[4] M. Haverbeke, Eloquent Javascript: A Modern Introduction to Programming, 3rd, No Starch Press, 2018.
[5] S. Souders, High performance web sites: essential knowledge for front-end engineers, O'Reilly Media, 2007.
[6] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, J.P. Singh, Who killed my battery?: Analyzing mobile browser energy consumption, in: Proceedings of the 21st international conference on World Wide Web, ACM, 2012, pp. 41–50.
[7] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, F. Tip, Efficient construction of approximate call graphs for javascript ide services, in: Proceedings of the 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 752–761.
[8] G. Richards, S. Lebresne, B. Burg, J. Vitek, An analysis of the dynamic behavior of javascript programs, in: Proceedings of the ACM Sigplan Notices, 45, ACM, 2010, pp. 1–12.
[9] P. Meenan, How fast is your website? Commun. ACM 56 (4) (2013) 49–55.
[10] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, Experimentation in Software Engineering., Springer, 2012.
[11] E. Andreasen, A. Feldthaus, S.H. Jensen, C.S. Jensen, P.A. Jonsson, M. Madsen, A. Møller, Improving tools for javascript programmers, in: Proceedings of International Workshop on Scripts to Programs. Beijing, China:[sn], 2012, pp. 67–82.
[12] S.H. Jensen, A. Møller, P. Thiemann, Type analysis for javascript, in: International Static Analysis Symposium, Springer, 2009, pp. 238–255.
[13] S.H. Jensen, A. Møller, P. Thiemann, Interprocedural analysis with lazy propagation, in: Proceedings of the International Static Analysis Symposium, Springer, 2010, pp. 320–339.
[14] E. Andreasen, A. Møller, Determinacy in static analysis for jquery, ACM SIGPLAN Notices 49 (10) (2014) 17–31.
[15] A. Guha, S. Krishnamurthi, T. Jim, Using static analysis for ajax intrusion detection, in: Proceedings of the 18th international conference on World wide web, ACM, 2009, pp. 561–570.
[16] S.H. Jensen, M. Madsen, A. Møller, Modeling the html dom and browser api in static analysis of javascript web applications, in: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ACM, 2011, pp. 59–69.
[17] S. Benschop, Efficient Approximate JavaScript Call Graph Construction, University of Groningen, 2014 Master's thesis.
[18] A.M. Fard, A. Mesbah, Jsnose: Detecting javascript code smells, in: Proceedings of the 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2013, pp. 116–125.

[22] http://rollupjs.org/.