Online Appendix: Preliminary Experiment on Lacuna Performance

1 Introduction

In this online appendix, we present a preliminary experiment we conducted to assess the performance of Lacuna with respect to its correctness, completeness, and accuracy in detecting JavaScript dead code across different third-party analysis techniques (also referred to analyzers) and their combinations. Below, we first present the design of this experiment and then the obtained results.

2 Experiment Design

We plan (and execute) our in-the-lab experiment by following the guidelines by Wohlin *et al.* [WRH+12].

2.1 Goal and Research Questions

The goal of the in-the-lab experiment, formalized by using the GQM template [BR88], is the following:

> ***Analyze*** *Lacuna* ***for the purpose of*** *evaluating its performance* ***with respect to*** *correctness, completeness, and accuracy of detected JavaScript dead functions* ***from the point of view of*** *both researchers and developers* ***in the context of*** *JavaScript web apps.*

To pursue the above-mentioned goal, we formulate and then investigate three Research Questions (RQs), one for each quality focus of our goal.

RQ1. What is the performance of Lacuna in terms of *correctness* of detected JavaScript dead functions?

With RQ1 we want to understand the extent to which JavaScript dead functions detected by Lacuna are actually dead. Answering this RQ is important since if Lacuna (wrongly) detected too many dead functions that are actually alive in a web app, the execution of that web app would be likely to produce unwanted behaviors in case of Optimization Levels 2 and 3, or too many lazy-loading calls in case of Optimization Level 1 (see the paper).

RQ2. What is the performance of Lacuna in terms of *completeness* of detected JavaScript dead functions?

RQ2 aims to understand to what extent Lacuna is capable of detecting *all* JavaScript dead function of a web app. If Lacuna missed too many dead functions, then the benefits in terms of run-time overhead could be negligible. In other words, it may not be worth using Lacuna.

RQ3. What is the performance of Lacuna in terms of *accuracy* of detected JavaScript dead functions?

Finally, with RQ3 we aim at understanding the extent to which JavaScript dead functions detected by Lacuna are both correct and complete. That is, accuracy takes into account both correctness and completeness of detected dead functions.

2.2 Subjects selection

To evaluate the performance of Lacuna, we run it on 39 JavaScript web apps, developed by independent web developers, belonging to the TodoMVC project.[1] This project aims to help developers to choose the MV* (Model View Anything) framework more suitable for structuring and organizing their JavaScript web apps. To that end, TodoMVC consists of different implementations of the same Todo app, each of which uses a different MV* framework, so that developers can inspect the codebase and then compare the different MV* frameworks. The Todo app is a manager for to-do lists, which includes the following features: (i) adding a to-do item, (ii) removing a to-do item, (ii) modifying an existing to-do item, and (iv) marking a to-do item as completed.

In Table 1, we report some information (i.e., total number of functions as parsed by Esprima, as well as number of alive and dead Java Script functions) about the web apps we selected for evaluating Lacuna. To distinguish each web app, we refer to it as the name of the used MV* framework. The choice behind these 39 web apps is driven by three main reasons. First, these web apps allow studying a *wide* and *heterogeneous* set of MV* frameworks that real-world JavaScript web apps can rely on. Second, they have a not-so-large number of functions so that we can profile and inspect the web apps easily. Third, they can be considered *representative of real-world web apps* because: (i) they use common JavaScript coding patterns (e.g., event handlers, local data storage, etc.), as well as common MV* frameworks for JavaScript web apps; (ii) they must comply with a rigorous set of requirements, coding styles, HTML/CSS templates, and other specifications [GA13]; and (iii) they are high-quality solutions since any web app is included in the TodoMVC project only after passing two reviews—the former performed by the leaders of the project, the latter performed by the open-source community [GA13].

2.3 Variables

Lacuna includes eight ready-to-use analyzers, which can be used alone, or combined with each other, to detect and then remove dead functions from JavaScript web apps. Depending on the analyzer/s Lacuna exploits to detect dead functions, we can distinguish among different instances of Lacuna. For example, there is an instance of Lacuna that exploits the ACG analyzers only, while another one exploits a combination of two analyzers like npm_cg and TAJS, and so on. Accordingly, the variable we manipulate in our experiment (i.e., the independent variable) is the *instance* of Lacuna. The number of possible instances is 255.

To quantify correctness (RQ1), completeness (RQ2), and accuracy (RQ3) of detected dead functions, we use the following dependent variables: *Precision* (P), *Recall* (R), and *F-score* (F) [MRS08]. Let x be an instance of Lacuna, P and R are defined as follows:

$$P(x) = \frac{|dead(x) \cap dead|}{|dead(x)|} \tag{1}$$

$$R(x) = \frac{|dead(x) \cap dead|}{|dead|} \tag{2}$$

where $dead(x)$ is the set of dead functions the Lacuna instance detected on a given web app, while $dead$ is the true set of dead functions (i.e., the ground truth) for that web app. Both P and R range in $[0, 1]$. The higher the P value, the more correct the detected dead methods are. On the other hand, the higher the R value, the more complete the set of detected dead methods is. As for F, it is defined as the harmonic mean of P and R, namely:

$$F(x) = \frac{2 * P * R}{P + R} \tag{3}$$

F ranges in $[0, 1]$ and a value close to 1 indicates high accuracy in detecting dead functions. P, R, and F are borrowed from the IR field where these measures are commonly used to assess correctness, completeness, and accuracy [MRS08].

[1] `https://todomvc.com/`

Table 1: Some information on the web apps selected for evaluating Lacuna.

web App	**#Functions**	**#Alive Functions**	**#Dead Functions**
ampersand	774	390	384
angularjs_require	127	80	47
ariatemplates	472	264	208
aurelia	357	246	111
backbone	950	409	541
backbone_require	127	88	39
canjs	1,105	459	646
canjs_require	128	76	52
closure	591	175	416
dijon	834	299	535
dojo	1,074	520	554
enyo_backbone	20	15	5
exoskeleton	256	102	154
flight	127	88	39
jquery	869	312	557
knockoutjs	560	235	325
knockoutjs_require	128	76	52
lavaca_require	981	303	678
mithril	136	52	84
olives	533	185	348
polymer	19	13	6
puremvc	229	121	108
ractive	932	431	501
rappidjs	867	405	462
react	1,848	944	904
react-alt	2,033	1,026	1,007
riotjs	170	125	45
sapui5	16	12	4
serenadejs	400	165	235
somajs	342	172	170
somajs_require	128	80	48
spine	999	245	754
troopjs_require	130	79	51
typescript-angular	1,440	609	831
typescript-backbone	1,243	464	779
typescript-react	1,105	674	431
vanilla-es6	80	69	11
vanillajs	131	98	33
vue	622	352	270

2.4 Experiment execution

To compute the values of the dependent variables, we need to know the true set of dead functions (i.e., the ground truth) of each web app. Inspired by past work [BHG12, EJJ^{+}12], we instrumented each function of all web apps with a logging statement in order to record whether that function is executed, or not, at run-time. To run each web app and thus record the execution of its functions, we leverage the already-existing test suite of each web app. In particular, each web app of the TodoMVC project comes with a test suite exercising all end points of that application. We use such a test suite to automatically run each web app, exercising all end points, and then record the executed functions. Any executed function is thus marked as alive. On the other hand, any non-executed function is marked as dead. The number of alive and dead functions of each selected web app is shown in Table 1.

Once obtained the true ground truth of each web app, we run each instance of Lacuna and gather the functions detected as dead. By comparing the true set of dead functions with those detected by each instance of Lacuna, for a given web app, we can compute the values of P, R, and F.

When executing the WALA analyzer alone on the selected web apps, it turned out to be too slow. In particular, the execution times of WALA were even greater than 20 minutes for most of the web apps, so making WALA unusable in a real context—the laptop used to execute the experiment was a MacBook Pro equipped with an Intel Core i7 processor (2,7GHz) and 16GB of RAM, and runs MacOS 10.14 as operating system. Therefore, we discard WALA (including any combination with other analyzers) from the experiment; as a consequence, the number of instances of Lacuna dropped to 127.

2.5 Data Analysis

To analyze the data and thus answer our research questions, we compute descriptive statistics like: min (minimum), max (maximum), median, mean, SD (Standard Deviation), and CV (Coefficient of Variation). These descriptive statistics allows us to understand the distributions of the values for each dependent variable and Lacuna instance.

3 Experiment Results

In this section, we report the results by research question.

3.1 Correctness of Detected Dead Functions (RQ1)

In Table 2, we show some descriptive statistics of the P values achieved by the instances of Lacuna based on single analyzers. We also show (in the lower part of Table 2,) the top three instances of Lacuna, ranked by average of P, based on combinations of analyzers. To identity a combination of analyzers, we used a sequence of letters where each letter identifies a single analyzer. For instance, let 'a' be the letter identifying ACG and 'd' be the one identifying Native Calls, 'ad' indicates the combination of ACG and Native Calls. The interested reader can find, in our replication package[2], the descriptive statistics of the P dependent variable (as well as R and F) for any combination of analyzers.

Table 2: Descriptive statistics for the P values.

Instance	**Min.**	**Max.**	**Median**	**Mean**	**SD**	**CV**
Instances based on single analyzers						
ACG (a)	0.212	0.900	0.630	0.621	0.159	25.541
Closure Compiler (b)	0.138	0.880	0.524	0.504	0.178	35.281
Dynamic (c)	0.208	0.992	0.870	0.819	0.189	23.038
Native Calls (d)	0.138	0.846	0.497	0.492	0.166	33.625
npm_cg (e)	0.208	0.885	0.519	0.522	0.152	29.189
Static (f)	0.189	0.898	0.600	0.599	0.180	30.121
TAJS (g)	0.149	0.864	0.503	0.504	0.160	31.665
Instances based on combinations of analyzers (only the top three by average of *P*)						
abcdeg	0.250	0.997	0.926	0.881	0.147	16.730
abceg	0.250	0.997	0.926	0.881	0.147	16.730
abcg	0.250	0.997	0.929	0.880	0.148	16.766

As shown in Table 2, there is a clear winner among the instances of Lacuna based on single analyzers, namely: Dynamic. Indeed, with the Dynamic analyzer, Lacuna reaches the highest P values on average (0.819) and median (0.870). That is, when using the Dynamic analyzer, 81.9% of the detected dead functions are correct on average; while 87% of the detected dead functions are correct on median. Dynamic also exhibits the lowest CV value (23.038), meaning that the level of dispersion around the mean is the lowest one.

After looking at the lower part of Table 2, it is easy to grasp that the combination of more analyzers in Lacuna results in a winning choice since we can observe improvements on all the descriptive statistics of the P values with respect to Dynamic (i.e., the best Lacuna instance using a single analyzer). For example, when combining all the analyzers except for Static (i.e., 'abcdeg'), the P value is equal to

[2]`https://github.com/Kishanjay/LacunaV2-evaluator`

0.881 on average and equal to 0.926 on median, so leading to an improvement, with respect to Dynamic, equals to 7.57% and 6.44% on mean and median, respectively. Moreover, a lower dispersion around the mean (CV = 16.730) is reached. Finally, we can observe that the difference in the distributions of the P values, among the top three instances of Lacuna based on multiple analyzers, is quite irrelevant (see any descriptive statistic in the lower part of Table 2).

Summing up, among the single analyzers, Dynamic allows achieving the best performance in terms of correctness of detected dead functions—on average, 81.9% of the functions Dynamic detects as dead are correct. By combining more analyzers, Lacuna can correctly detect up to 88.1% of dead methods on average.

3.2 Completeness of Detected Dead Functions (RQ2)

Table 3 reports some descriptive statistics of the R values achieved by the instances of Lacuna based on single analyzers, on the top, and on combinations of analyzers (limited to the top three combinations ranked by average of R), on the bottom.

Table 3: Descriptive statistics for the R values.

Instance	**Min.**	**Max.**	**Median**	**Mean**	**SD**	**CV**
Instances based on single analyzers						
ACG (a)	0.324	1.000	0.569	0.602	0.191	31.772
Closure Compiler (b)	0.434	1.000	1.000	0.863	0.182	21.070
Dynamic (c)	0.696	1.000	1.000	0.979	0.058	5.943
Native Calls (d)	0.991	1.000	1.000	1.000	0.001	0.147
npm_cg (e)	0.622	1.000	0.928	0.890	0.094	10.560
Static (f)	0.315	0.997	0.549	0.594	0.185	31.128
TAJS (g)	0.800	1.000	1.000	0.993	0.032	3.248
Instances based on combinations of analyzers (only the top three by average of R)						
cd	0.696	1.000	1.000	0.979	0.058	5.943
cg	0.688	1.000	1.000	0.972	0.065	6.721
dg	0.800	1.000	1.000	0.993	0.032	3.247

The results in Table 3 points out that the best instance of Lacuna, among those based on single analyzers, is Native Calls followed by TAJS and Dynamic. For these three instances, we can observe an R value equal to 1 on median, and equal to or very close to 1 on average. Moreover, the levels of dispersion around the means are the lowest ones (CV values equal to 0.147, 3.248, 5.943 for Native Calls, TAJS and Dynamic, respectively).

If we consider the top three combinations of analyzers (see the lower part of Table 3), ranked by average of R, we can notice that these combinations result from selecting two analyzers among Dynamic, Native Calls, and TAJS (i.e., the best single analyzers). Moreover, by looking at any descriptive statistic in Table 3, we can notice that there is not a great improvement in the R values when Lacuna uses these three combinations of analyzers as compared to the use of only Dynamic, Native Calls, or TAJS.

In summary, Lacuna can detect, on average, from 97.2% to 100% of all dead functions available in a web app if it use Dynamic, Native Calls, or TAJS as a single analyzer, or when using a combination of two analyzers among those aforementioned.

3.3 Accuracy of Detected Dead Functions (RQ3)

In Table 4, we show some descriptive statistics of the F values achieved by the instances of Lacuna based on single analyzers and on combinations thereof (only the top thee instances ranked by average of P).

By looking at Table 4, it is easy to identify a winner, among the instances of Lacuna based on single analyzers, in terms of accuracy of detected dead functions, namely: Dynamic. Indeed, Lacuna with the Dynamic analyzer reaches the highest F values on average (0.877) and median (0.918). In light of the results on correctness and completeness of detected dead functions (see Table 2 and Table 3), this is not

Table 4: Descriptive statistics for the F values.

Instance	**Min.**	**Max.**	**Median**	**Mean**	**SD**	**CV**
Instances based on single analyzers						
ACG (a)	0.318	0.773	0.583	0.577	0.105	18.147
Closure Compiler (b)	0.242	0.826	0.613	0.600	0.136	22.603
Dynamic (c)	0.344	0.996	0.918	0.877	0.142	16.137
Native Calls (d)	0.242	0.916	0.664	0.643	0.155	24.176
npm_cg (e)	0.344	0.854	0.650	0.642	0.124	19.280
Static (f)	0.292	0.841	0.555	0.557	0.100	17.874
TAJS (g)	0.259	0.921	0.669	0.654	0.147	22.402
Instances based on combinations of analyzers (only the top three by average of *F*)						
cd	0.344	0.996	0.918	0.877	0.142	16.137
cg	0.344	0.996	0.918	0.879	0.138	15.655
cdg	0.344	0.996	0.918	0.879	0.138	15.655

a surprising outcome. That is, Lacuna with the Dynamic analyzer achieved the best performance in terms of correctness of detected dead functions by showing a very high level of completeness; therefore, we could expect that Dynamic allowed Lacuna to reach the best performance in terms of accuracy (which balances correctness and completeness). It is also worth noting that Dynamic leads to more stable F values when passing from a web app to another one. This is because Dynamic has the lowest CV value (16.137).

As for the instances of Lacuna based on combinations of analyzers (see the lower part of Table 4), the best instances are two: the one using Dynamic and TAJS; and the one using Dynamic, Native Calls and TAJS. For both these instances, the F values are equal to 0.879 on average and 0.918 on median (see also the other descriptive statistics in Table 4). In a nutshell, combining Dynamic and TAJS or Dynamic, Native Calls and TAJS lead to a small improvement in the F values with respect to Dynamic.

We can conclude that Lacuna can reach, on average, an accuracy level of detected dead functions equal to 87.7%, if is uses the Dynamic analyzer, and up to 87.9%, if it uses a combination of Dynamic and TAJS (or a combination of Dynamic, TAJS, and Native Calls). If we had to choose a Lacuna instance (out of 127 possible instances), our choice would be the one based on the joint use of Dynamic and TAJS. This is because such instance allows achieving the highest accuracy level (so well balancing correctness[3] and completeness) and, at the same time, exploits fewer analyzers than the Lacuna instance based on the combination of Dynamic, TAJS, and Native Calls.

References

[BHG12] H. Boomsma, B. V. Hostnet, and H. G. Gross. Dead code elimination for web systems written in php: Lessons learned from an industry case. In *Proceedings of the 28th International Conference on Software Maintenance*, pages 511–515. IEEE, 2012.

[BR88] V. R. Basili and H. D. Rombach. The tame project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, 1988.

[EJJ+12] Sebastian Eder, Maximilian Junker, Elmar Jurgens, Benedikt Hauptmann, Rudolf Vaas, and Karl-Heinz Prommer. How much does unused code matter for maintenance? In *2012 34th International Conference on Software Engineering*, pages 1102–1111. IEEE, 2012.

[GA13] 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.

[3]The descriptive statistics (not reported in Table 2) for the P values resulting from the combination of Dynamic and TAJS are: min = 0.208, max = 0.992, median = 0.870, mean = 0.825, SD = 0.181, and CV = 21.988.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schutze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[WRH$^+$12] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.