Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

An Empirical Study on the Run-time Impact of JavaScript Dead Code

Author: Khalil Mayyasi (2644550)

1st supervisor:	Ivano Malavolta
daily supervisor:	Ivano Malavolta
2nd reader:	Emitza Guzman

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

January 18, 2023

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

JavaScript is a programming language used in most current web applications, it is used to implement a more interactive web application. Nowadays, interactive web applications are highly desired. Therefore, JavaScript frameworks and libraries are used to decrease the duration of the development cycle for implementing interactive web applications. Web applications usually adopt few functionalities from a framework/library, the unused functionalities are known as dead code. These functionalities are never executed, however, are downloaded and parsed which results in wasting several assets.

The goal of my project is to understand the impact of the presence of JavaScript dead code on energy consumption of mobile device, on various performance metrics of the mobile web applications including memory consumption, CPU usage, loading time, first paint and first contentful paint and on various network usage metrics including packets transferred and bytes transferred.

To understand the impact of JavaScript dead code, we select 36 web apps and removed JavaScript dead code using three different methods. Then, we perform an experiment to run all four versions of each subjects while measuring the metrics selected.

The result of our experiment shows that energy consumption, loading time, CPU usage, packets and bytes transferred are negatively impacted by the presence of JavaScript dead code. However, the presence of JavaScript dead code has no impact on the metrics memory consumption,fp and fcp. Furthermore, Dunn's test provides statistical results showing the significant treatments for each metrics when compared to the original treatment. In conclusion, we recommend developers to use the results to understand that while developing a web application they should use and import only the wanted functions from a JavaScript framework/library. Otherwise, the presence of unused functions will negatively impact various run-time properties of the web application.

Contents

List of Figures iii

List of Tables v

1 Introduction 1

2 Structure of the thesis 3

3 Background 5

3.1 JavaScript dead code 5

3.2 LacunaV2 Tool 6

4 Experiment Definition 9

4.1 Goal 9

4.2 Questions 10

4.3 Metrics 10

4.3.1 Energy Consumption 11

4.3.2 Performance 11

4.3.2.1 CPU usage 11

4.3.2.2 Memory Consumption 11

4.3.2.3 Loading Time 11

4.3.2.4 First Paint (FP) 11

4.3.2.5 First Contentful Paint (FCP) 12

4.3.3 Network Usage 12

4.3.3.1 Packets Transferred 12

4.3.3.2 Bytes Transferred 12

4.4 GQM-Tree 12

CONTENTS

5 Experiment Planning **15**
5.1 Subjects Selection . . . 15
5.2 Experimental Variables . . . 21
5.3 Experimental Hypotheses . . . 22
5.4 Experiment Design . . . 24

6 Experiment Execution **27**
6.1 Setup & Execution . . . 27

7 Results **33**
7.1 Result Representation . . . 33
7.2 Normality Check . . . 41
7.3 Hypothesis Testing . . . 43

8 Discussion **49**

9 Threats To Validity **53**
9.1 External Validity . . . 53
9.2 Internal Validity . . . 53
9.3 Construct Validity . . . 54
9.4 Conclusion Validity . . . 54

10 Related Work **55**

11 Conclusion and Future Work **57**

12 Appendix **59**

References **77**

List of Figures

4.1 GQM-Tree 13

6.1 Experiment Setup Illustration 28

7.1 Violin plot of CPU usage across the treatments 33
7.2 Violin plot of energy consumption across the treatments 34
7.3 Violin plot of memory consumption across the treatments 34
7.4 Violin plot of FCP across the treatments 35
7.5 Violin plot of FP across the treatments 35
7.6 Violin plot of loading times across the treatments 36
7.7 Violin plot of kilobytes transferred across the treatments 36
7.8 Violin plots of packets transferred across the treatments 37
7.9 Dunn test for energy consumption 44
7.10 Dunn test for CPU usage 44
7.11 Dunn test for memory consumption 45
7.12 Dunn test for loading time 46
7.13 Dunn test for FP 46
7.14 Dunn test for FCP 47
7.15 Dunn test for packets transferred 48
7.16 Dunn test for bytes transferred 48

12.1 Normal QQ plot of OL0 Energy consumption 59
12.2 Normal QQ plot of OL1 Energy consumption 60
12.3 Normal QQ plot of OL2 Energy consumption 60
12.4 Normal QQ plot of OL3 Energy consumption 61
12.5 Normal QQ plot of OL0 CPU Usage 61
12.6 Normal QQ plot of OL1 CPU Usage 62

12.7 Normal QQ plot of OL2 CPU Usage 62
12.8 Normal QQ plot of OL3 CPU Usage 63
12.9 Normal QQ plot of OL0 Memory consumption 63
12.10Normal QQ plot of OL1 Memory consumption 64
12.11Normal QQ plot of OL2 Memory consumption 64
12.12Normal QQ plot of OL3 Memory consumption 65
12.13Normal QQ plot of OL0 Loading Time 65
12.14Normal QQ plot of OL1 Loading Time 66
12.15Normal QQ plot of OL2 Loading Time 66
12.16Normal QQ plot of OL3 Loading Time 67
12.17Normal QQ plot of OL0 FP 67
12.18Normal QQ plot of OL1 FP 68
12.19Normal QQ plot of OL2 FP 68
12.20Normal QQ plot of OL3 FP 69
12.21Normal QQ plot of OL0 FCP 69
12.22Normal QQ plot of OL1 FCP 70
12.23Normal QQ plot of OL2 FCP 70
12.24Normal QQ plot of OL3 FCP 71
12.25Normal QQ plot of OL0 Packets transferred 71
12.26Normal QQ plot of OL1 Packets transferred 72
12.27Normal QQ plot of OL2 Packets transferred 72
12.28Normal QQ plot of OL3 Packets transferred 73
12.29Normal QQ plot of OL0 Bytes transferred 73
12.30Normal QQ plot of OL1 Bytes transferred 74
12.31Normal QQ plot of OL2 Bytes transferred 74
12.32Normal QQ plot of OL3 Bytes transferred 75

List of Tables

4.1 Goal Definition 9

5.1 Demographics for subjects in OL-0 17
5.2 Demographics for subjects in OL-1 18
5.3 Demographics for subjects in OL-2 19
5.4 Demographics for subjects in OL-3 20
5.5 Experiment Variables 21

7.1 Overview of CPU usage values 38
7.2 Overview of energy consumption values 38
7.3 Overview of Memory consumption values 38
7.4 Overview of FCP values 38
7.5 Overview of FP values 39
7.6 Overview of Loading Time values 39
7.7 Overview of Bytes transferred values (in KB) 39
7.8 Overview of packets transferred values 39
7.9 Shapiro-Wilk normality test results 42

LIST OF TABLES

1

Introduction

Web application front-ends are usually constructed using HTML, CSS, and JavaScript. JavaScript is a programming language used to implement a more interactive web applications. Recently, the need for high interactive web applications has increased. Therefore, many developers are using frameworks and libraries to help build large scale interactive web applications. JavaScript frameworks/libraries (e.g. jQuery and Angular) can be also used to help in the development of mobile web applications.(1)(2).

JavaScript frameworks/library allow developers to be more efficient by being able to reuse code that is already developed and tested thoroughly. However, with the positives of using frameworks/libraries comes some negative repercussions, when a web application uses a JavaScript framework/library not all functionalities are used. Therefore, the unused functions in the framework/library also known as ***dead code*** are never executed at runtime of the web application, however they are still downloaded and parsed by the JavaScript engine(3).

The presence of JavaScript dead code has no impact on functionalities of the web application, however it increases the size of the web application (4). We hypothesize that this increase can impact various metrics such as energy consumption, memory consumption, loading time, render time, packets and bytes transferred of the web application as the code is still downloaded and parsed. Therefore, the presence of JavaScript dead code results in wasting various assets.

The **goal** of this paper is to inspect the impact of JavaScript dead code on the energy consumption, performance and network usage metrics of mobile web applications. To achieve the goal of the paper, we design and conduct an experiment where four variants of each of the 36 mobile web applications(16 real mobile web applications and 20 in-the-lab mobile web applications) are analyzed in terms of their energy consumption, performance

and network usage. The four variants are generated using different JavaScript dead code removal approaches. After the experiment is executed and the measurements are collected for the 20 runs of each variant of each subject, we statistically analyze if the four variants of the subjects are impacted differently in terms of energy consumption, performance and network usage.

The **results** of the experiment reveal that the energy consumption and the network usage metrics are negatively impacted by the presence of JavaScript dead code. However, the results of the performance metrics reveal that only the CPU usage and loading time from the performance metric are negatively impacted by the presence of JavaScript dead code. The render timing variables in the performance metric are not impacted by the presence of JavaScript dead code.

The **target audience** of this paper are developers whose main aim is developing mobile web applications. This paper supports them by providing evidence of the impact of JavaScript dead code on energy consumption, performance and network usage metrics of mobile web applications running on Android devices. Therefore, when a mobile web application is being developed, the outcome of the experiment will help developers understand which run-time properties of the mobile web application are impacted if the developers incorporate dead code or unused functions in the mobile web applications.

The **main contribution** of this paper are (i) an empirical study on the run-time impact of JavaScript dead code on the energy consumption, performance and network usage of mobile web applications; (ii) a discussion of the results of the experiment from the perspective of mobile web application developers; (iii) a replication package of the experiment containing the results, all variants of each subject, scripts used to execute the statistical tests, the tools used to collect the measurements, and the tool used to execute the experiment[1].

[1]https://github.com/KhalilMay/Master-JavaScript-Dead-Code

2

Structure of the thesis

The structure of thesis for the following chapters is as follows, *Chapter 3* includes a background description of JavaScript dead code and a description of the tool used to detect and remove JavaScript dead code. *Chapter 4* describes the goal of our experiment and research question we are aiming to solve. *Chapter 5* describe the plan process of the experiment and the experimental variables involved. *Chapter 6* provides information for the experiment setup and execution including the environment and tools involved in the experiment. *Chapter 7* provides various plots and hypothesis testing to outline the results of the experiment. *Chapters 8* includes a discussion of the obtained results of the experiment. *Chapter 9* discusses the possible threats to validity of the paper. *Chapter 10* includes related work. *Chapter 11* consist of conclusion of the experiment and potential future work.

2. STRUCTURE OF THE THESIS

3

Background

3.1 JavaScript dead code

JavaScript dead code is the unused JavaScript code/functions that are incorporated in the code of web applications but never executed. JavaScript dead code can cause several negative impacts on the web application. These impacts include unnecessary increased of the size of JavaScript files which also increases the network transfer time of the files, increased parsing overhead on the JavaScript engine, and increased loading time and energy consumption for downloading web applications with dead code(5). Many developers have tried to build tools to help improve JavaScript code such as JSNose[1]. However, those tools are not able to detect and eliminate JavaScript dead code at the function level(6).

JavaScript dead code can be present in numerous types throughout the code. Listing 3.1 provides two examples of JavaScript dead code. Line 1 shows an import for a library, from which limited number of functions are used. However, still all functions of the library is loaded to the client whenever this code is executed. The function "uncalledFunctionName" contains a console.log statement at line 5 which is never executed since the function is never called. Therefore, the function and its content are declared as dead code as they are never executed.

[1] http://salt.ece.ubc.ca/content/jsnose/

Listing 3.1: JavaScript dead code Examples

```
1 import * as frameworkLibrary from "frameworkLibrary-name";
2
3
4 function uncalledFunctionName(){
5   console.log("Code Never Reached"); //unreached code (Dead Code)
6 }
```

3.2 LacunaV2 Tool

LacunaV2 is an analysis tool used to detect and remove dead code at the function level using a call graph. It includes several analyzers that detect dead code in their distinct approach. Those analyzers can be used together to improve the accuracy of dead code detection. A list of the analyzers available in the tool are shown below(7).

- **Static analyzer**: performs basic static analysis to generate a call graph.
- **Dynamic analyzer**: loads the entry file and marks all executed functions as alive.
- **Nativecalls** analyzer: only considers calls from and to native JavaScript functions.
- **WALA analyzer**: is an extension of the WALA static analysis framework. The analyzer builds an intermediate form of the JavaScript of being analyzed.
- **ACG analyzer**: is a field-based call graph construction for JavaScript.
- **TAJS analyzer**: is a data-flow analysis for JavaScript that infers type information and call graphs.
- **npm_cg analyzer**: produces call graphs from JavaScript source code.
- **ClosureCompiler analyzer**: creates call graphs to compile JavaScript code to a better JavaScript code by parsing. analyzing the code and removing dead code.

Furthermore, the tool has 4 optimization levels, with each optimization level using different mechanism to remove JavaScript dead code. However, optimization level-0 does not remove any dead code but it is used to help identify which functions are dead without actually removing them. Additionally, this level can be also used to compare the measurements collected from the other levels.

LacunaV2 is an expansion of the original Lacuna tool(1). LacunaV2 is a more robust and user-friendly tool compared to the original Lacuna tool. The robustness is improved by fixing problems occurring in the original tool, these fixes includes introducing error handling for web apps that produces errors at run-time. Additionally, the tool includes the ability for the user to select the elimination strategy to use, therefore improving the user-friendliness of the tool.

4

Experiment Definition

To define our experiment we will be describing the goal of the experiment, the research questions that we are want to answer, and the metrics that are measured during the experiment execution.

4.1 Goal

The goal of the experiment is constructed using the Goal-Question-Metric Approach (GQM)(8)(9). The goal is to analyze JavaScript dead code for the purpose of evaluating its impact on energy consumption, performance and network usage of mobile web applications from the point view of software developers. This goal is also shown in Table 4.1.

Table 4.1: Goal Definition

Analyze	JavaScript dead code
For the purpose of	Evaluating its impact
With respect to their	Energy Consumption, Performance and Network usage
From the point of view of	Software Developers
In the context of	Mobile web applications

4.2 Questions

We constructed three research questions from the goal of the experiment described earlier.

Research Question 1:

What is the impact of JavaScript dead code on the energy consumption of mobile web applications?

For RQ1, we track the energy consumption of the mobile device for each web application in every optimization level. Comparing optimization level-0 to the other optimization levels, we will be able to understand how JavaScript dead code impacts the energy consumption of the device.

Research Question 2:

What is the impact of JavaScript dead code on the performance of mobile web applications?

For RQ2, we track performance measurements for all optimization levels of the experiment subjects. Similarly to RQ1, comparing optimization level-0 to the other optimization levels, we will be able to understand how JavaScript dead code impacts the performance of the mobile web apps.

Research Question 3:

What is the impact of JavaScript dead code on the network usage of mobile web applications?

For RQ3, we track network usage measurements for all optimization levels of the experiment subjects. Similarly to RQ1 & RQ2, comparing optimization level-0 to the other optimization levels, we will be able to understand how JavaScript dead code impacts the network usage of the mobile web apps.

4.3 Metrics

We will be using three metrics in this experiment, each metric will be related to a single research question that we mentioned earlier and their measurements in the experiment will allow us to answer these questions.

4.3.1 Energy Consumption

To measure how much energy the battery is using to answer RQ1, we will be tracking the energy consumption. This metric will be measured in Joules(J). This metric is naturally collected by the AndroidRunner tool.

4.3.2 Performance

This metric will be analyzed by tracking several performance measurements listed below.

4.3.2.1 CPU usage

CPU usage is how much of the processor the device is using to execute the experiment and it is measured in percentage(%). The metric is collected using a script that uses *dumpsys* written within the AndroidRunner tool [1].

4.3.2.2 Memory Consumption

Memory consumption is how much memory the device consumes during the experiment runs. This metric is measured in MegaByte. Similarly to the energy consumption metric, this metric is naturally collected by the AndroidRunner tool.

4.3.2.3 Loading Time

Loading Time is the amount of time it takes the browser to fully load a specific subject, and it is measured in milliseconds (ms). To collect this metric, we inject a JavaScript code to each subject that measures the loading time. Then, the code transmit the result to a server that collects the result and writes it to a file.

4.3.2.4 First Paint (FP)

First Paint is the time it takes for the browser to render any visual aspect different from what was on screen before the navigation to the web page, and it is measured in milliseconds (ms). To collect this metric, we use an object of the perfume library in same injected JavaScript code we used to collect the loading time[2].

[1] https://developer.android.com/studio/command-line/dumpsys
[2] https://zizzamia.github.io/perfume/

4.3.2.5 First Contentful Paint (FCP)

First Contentful Paint is the time it takes for the browser to render the first bit of content from the DOM, and it is measured in milliseconds (*ms*). Similarly to *FP*, the FCP metric is also collected using the object of the perfume library injected in the JavaScript code.

4.3.3 Network Usage

This metric is analyzed by tracking two network usage measurements listed below.

4.3.3.1 Packets Transferred

Packet Transferred is the total amount of packets transferred from the server to the mobile device to load the mobile web application, and it is measured in number of packets. To collect this metric, we use mitmproxy. *Mitmproxy* is an open source interactive HTTPS intercepting proxy[1]. It can be used to intercept, inspect and modify web traffic.

4.3.3.2 Bytes Transferred

Bytes Transferred is the total amount of Bytes transferred from the server to the mobile device to load the mobile web application, and it is measured in bytes. Similarly to *packets transferred*, we use mitmproxy to collect the number bytes transferred.

4.4 GQM-Tree

Since we have the goal, question and metrics constructed, we use them to create the GQM-tree in Figure 4.1.

[1]https://github.com/mitmproxy/mitmproxy

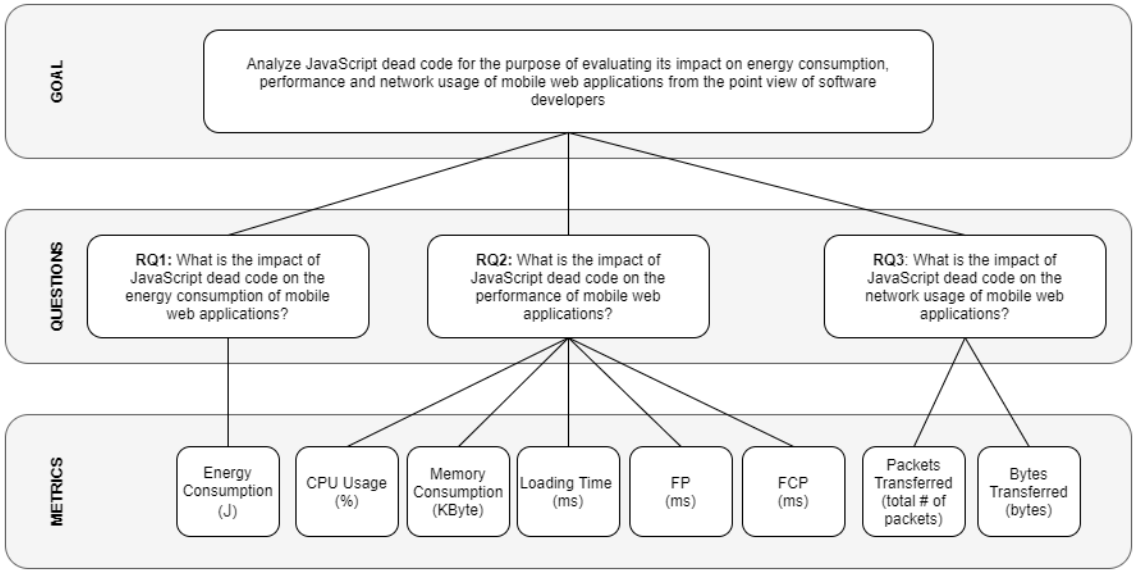


Figure 4.1: GQM-Tree

5

Experiment Planning

For the plan of the experiment, we will specify the subject selection. Furthermore, we will illustrate the experimental variables used, the hypothesis used for reporting the results and the design in which the experiment is executed.

5.1 Subjects Selection

For the subjects of our experiment, we use two different sets. The reason for including two sets is because for first set of subjects we will look at web pages that are being developed and has not been changed several times. However, the second set of subjects are the web apps which are already published and have been used for a long time and the code has been updated several times. Therefore, we want to include two sets of web apps to ensure our experiment results addresses a variety of web apps. The first set is for *real mobile web application experiment* which includes 16 web apps collected from a previous project about JavaScript dead code removal[1](10). This project used Tranco list to select the subjects and they had 3 inclusion criteria for subject selection which were: 1) LacunaV2 is able to apply the removal of dead code in the web applications; 2) The mobile web app must be accessible without the need to login; 3) The mobile web app should not redirect to another domain. The second set of subjects is for *in-the-lab experiment* which includes 20 web apps collected from TodoMVC collection, which are also selected based on if they are able to run on LacunaV2[2].

We are using two separate experiments in-the-lab and real mobile web applications, however, both experiments are executed under the same environment and same hardware

[1]https://github.com/GreenLab-Kebab/gl-kebab-subjects/tree/master/lacunaWebPages
[2]https://github.com/NielsGrootObbink/todomvc/tree/master/examples

instruments. Furthermore, both sets use the same analyzers combination of dyn+tajs for JavaScript dead code removal. *Kishan Nirghin* paper calculates the precision and recall of all possible analyzer combinations when detecting JavaScript dead code(7). These two metrics are used to calculate the f1score for each combination, which the results show that dyn+tajs is one of the combinations with the highest f1score. Tables 5.1, 5.2, 5.3, 5.4 display different demographics of the subjects showing the impact of the different optimization levels. The demographics within the tables include the number of *lines of code(LOC)* for JavaScript, the size of each file in the web app(CSS, JS, HTML), the number of JS functions, total LOC of JS functions, and the number of files for CSS, HTML and JS. To collect these demographics we use several tools and commands. For LOC, we use a tool called **Count Lines of Code(Cloc)**[1]. Additionally, we use the **Stat**[2] command in the terminal to collect various file sizes in bytes(B) within the subjects. Finally, we use the library **Esprima**[3] to write a JS code to detect and count all JS functions within a web app including their LOC[4].

The demographics tables show that the values are decreasing in tables OL0 through OL3. However, the number of files generally did not change which is due to the dead code removal process eliminating dead code within JavaScript files and not eliminating the files. Furthermore, some web pages such as *angularjs_require* shows the total *JSFunct-Loc* to be higher than *JS-Loc*. The reason for these irregular cases is because the *Cloc* command counts the line of codes within each file. However, some JavaScript files contain multiple lines of code written within one line, which Cloc considers as one line rather than counting the multiple lines of code.

[1]https://github.com/AlDanial/cloc
[2]shttps://linuxize.com/post/stat-command-in-linux/
[3]https://esprima.org/index.html
[4]https://github.com/KhalilMay/Master-JavaScript-Dead-Code/tree/master/CodeForDemographics

Table 5.1: Demographics for subjects in OL-0

Subjects	JS-Loc	JSFunc	JSFunc-Loc	JS(B)	CSS(B)	HTML(B)	JS Files	CSS Files	HTML Files
Real mobile web applications (Optimization Level 0)									
apache.org	4418	736	10411	205112	151178	65611	7	2	1
aws.amazon.com	10724	1302	27222	414936	0	170229	11	0	1
m.youtube.com	68999	10490	147719	2443831	609457	52191	12	2	1
nl.godaddy.com	3489	63	4746	219155	732827	143599	19	3	3
stackexchange.com	4306	805	9664	174246	0	110573	10	0	1
stackoverflow.com	5198	922	11322	215730	0	89317	8	0	1
www.amazon.com	4320	596	8746	329400	0	83180	57	0	4
www.amazon.in	10061	1452	26697	458073	0	419260	89	0	11
www.bbc.com	5583	961	12246	239196	0	132489	62	0	3
www.booking.com	25355	3787	52672	1225294	0	168810	32	0	4
www.buzzfeed.com	68144	11477	189013	3488298	161525	567398	21	1	4
www.mozilla.org	4463	881	9775	197519	0	149386	6	0	1
www.office.com	7891	1295	17096	319609	0	93696	10	0	2
www.paypal.com	10825	1246	23150	383174	0	75540	14	0	1
www.theguardian.com	3472	95	1337	180451	0	1145634	6	0	2
www.wikipedia.org	900	147	1457	43825	0	104402	5	0	1
In-the-lab (Optimization Level 0)									
angularjs_require	11556	1354	67284	1049617	8356	3129	12	2	1
backbone	8961	950	28151	396300	8754	2666	11	2	1
backbone_require	10299	1037	33863	479050	8356	2049	14	2	3
canjs	11317	1105	42300	573104	8754	2917	7	2	1
canjs_require	12560	1213	47634	754297	8754	2689	36	2	1
dijon	8957	834	31439	375266	8754	2488	12	2	1
dojo	1930	1294	5178	251974	8754	5939	26	2	3
enyo_backbone	13676	2483	17997	476607	8356	1295	28	2	1
exoskeleton	1876	256	4868	98544	8754	2741	11	2	1
gwt	3949	5419	4325	534879	12069	902	7	3	1
jquery	9477	869	32620	380624	8391	2621	5	3	1
jsblocks	11323	885	53229	460058	8754	2843	3	2	1
knockback	15979	1798	68127	782673	8754	3619	11	2	1
knockoutjs_require	1690	612	7249	150822	8356	2590	8	2	1
mithril	1431	136	3510	58721	8754	1489	8	2	1
polymer	205	1310	575	199723	8523	105624	4	2	15
reagent	53947	6154	117010	2580518	8789	979	2	2	1
vanilla-es6	596	234	895	47727	8789	1761	9	2	1
vanillajs	1040	182	3214	42007	8356	2613	9	2	2
vue	7664	622	19106	253574	8356	2592	6	2	1

Table 5.2: Demographics for subjects in OL-1

Subjects	JS-Loc	JSFunc	JSFunc-Loc	JS(B)	CSS(B)	HTML(B)	JS Files	CSS Files	HTML Files
Real mobile web applications (Optimization Level 1)									
apache.org	1574	571	2768	113495	151178	65611	7	2	1
aws.amazon.com	8358	1201	20437	368040	0	170229	11	0	1
m.youtube.com	51149	8290	95732	2008478	609457	52191	12	2	1
nl.godaddy.com	3329	61	3922	211760	732827	143599	19	3	3
stackexchange.com	1912	634	3237	119240	0	110573	10	0	1
stackoverflow.com	2080	687	3215	131224	0	89317	8	0	1
www.amazon.com	3706	581	6998	310331	0	83180	57	0	4
www.amazon.in	9534	1454	24760	441748	0	419260	89	0	11
www.bbc.com	4148	856	7635	213947	0	132489	62	0	3
www.booking.com	11095	2308	15279	683756	0	168810	32	0	4
www.buzzfeed.com	65713	11308	182039	3440666	161525	567398	21	1	4
www.mozilla.org	2214	723	3836	140594	0	149386	6	0	1
www.office.com	4360	1098	7967	233786	0	93696	10	0	2
www.paypal.com	5606	1050	9725	275176	0	75540	14	0	1
www.theguardian.com	3395	94	1111	177578	0	1145634	6	0	2
www.wikipedia.org	655	139	993	38382	0	104402	5	0	1
In-the-lab (Optimization Level 1)									
angularjs_require	11165	1334	65415	1032765	8356	3129	13	2	1
backbone	3643	797	10649	257387	8754	2666	12	2	1
backbone_require	9983	1026	32421	467601	8356	2049	15	2	3
canjs	6320	952	23722	420088	8754	2917	8	2	1
canjs_require	12151	1191	45719	737536	8754	2689	37	2	1
dijon	4455	712	15248	254316	8754	2488	13	2	1
dojo	1817	1142	4780	215796	8754	5939	27	2	3
enyo_backbone	13554	2479	17591	472812	8356	1295	29	2	1
exoskeleton	1077	237	2676	82612	8754	2743	13	2	1
gwt	3836	5416	3927	531773	12069	902	8	3	1
jquery	4759	734	15935	256395	8391	2621	6	3	1
jsblocks	7574	806	35953	381964	8754	2843	4	2	1
knockback	6604	1471	29731	493698	8754	3619	12	2	1
knockoutjs_require	1282	590	5310	133638	8356	2590	9	2	1
mithril	829	108	1769	45431	8754	1489	9	2	1
polymer	91	1306	174	196364	8523	105159	6	2	15
reagent	53823	6149	116597	2576658	8789	991	4	2	1
vanilla-es6	479	204	489	40690	8789	1761	10	2	1
vanillajs	711	167	2133	38067	8356	2613	10	2	2
vue	4874	561	11724	192239	8356	2592	7	2	1

Table 5.3: Demographics for subjects in OL-2

Subjects	JS-Loc	JSFunc	JSFunc-Loc	JS(B)	CSS(B)	HTML(B)	JS Files	CSS Files	HTML Files
Real mobile web applications (Optimization Level 2)									
apache.org	1547	568	2744	73363	151178	65611	7	2	1
aws.amazon.com	8313	1196	20397	325274	0	170229	11	0	1
m.youtube.com	51095	8284	95684	1802597	609457	52191	12	2	1
nl.godaddy.com	3302	58	3898	208864	732827	143599	19	3	3
stackexchange.com	1861	628	3195	71534	0	110573	10	0	1
stackoverflow.com	2017	680	3159	79645	0	89317	8	0	1
www.amazon.com	3607	570	6910	292192	0	83180	57	0	4
www.amazon.in	9288	1427	24540	417463	0	419260	88	0	11
www.bbc.com	3977	837	7483	173338	0	132489	62	0	3
www.booking.com	10951	2292	15151	537961	0	168810	32	0	4
www.buzzfeed.com	65668	11303	181999	3403430	161525	567398	21	1	4
www.mozilla.org	2178	719	3804	95517	0	149386	6	0	1
www.office.com	4306	1092	7919	169992	0	93696	10	0	2
www.paypal.com	5552	1044	9677	209166	0	75540	14	0	1
www.theguardian.com	3377	92	1095	175289	0	1145634	6	0	2
www.wikipedia.org	637	137	977	32261	0	104402	5	0	1
In-the-lab (Optimization Level 2)									
angularjs_require	11147	1332	65399	1028815	8356	3129	13	2	1
backbone	3568	788	10598	198030	8754	2666	12	2	1
backbone_require	9966	1025	32405	463979	8356	1659	16	2	3
canjs	6272	946	23695	367313	8754	2917	8	2	1
canjs_require	12119	1189	45634	732312	8754	2689	37	2	1
dijon	4374	703	15176	208478	8754	2488	13	2	1
dojo	1799	1140	4764	178535	8754	5939	27	2	3
enyo_backbone	13545	2478	17583	471856	8356	1295	29	2	1
exoskeleton	1031	228	2690	66297	8754	2743	13	2	1
gwt	3818	5414	3911	529448	12069	902	8	3	1
jquery	4714	729	15895	210821	8391	2621	6	3	1
jsblocks	7444	796	35411	331983	8754	2899	6	2	1
knockback	6523	1462	29659	389887	8754	3619	12	2	1
knockoutjs_require	1264	588	5294	129384	8356	2590	9	2	1
mithril	815	101	1829	39138	8754	1489	9	2	1
polymer	82	1305	166	195384	8523	105159	6	2	15
reagent	20669	4789	44344	1383755	8789	991	4	2	1
vanilla-es6	461	202	473	29645	8789	1761	10	2	1
vanillajs	648	160	2077	30844	8356	2613	10	2	2
vue	4820	555	11676	164123	8356	2592	7	2	1

Table 5.4: Demographics for subjects in OL-3

Subjects	JS-Loc	JSFunc	JSFunc-Loc	JS(B)	CSS(B)	HTML(B)	JS Files	CSS Files	HTML Files
Real mobile web applications (Optimization Level 3)									
apache.org	1501	131	2744	67600	151178	65611	7	2	1
aws.amazon.com	8277	787	20397	319969	0	170341	11	0	1
m.youtube.com	50324	6472	95682	1773924	609457	52191	12	2	1
nl.godaddy.com	3297	39	3898	208431	732827	143599	19	3	3
stackexchange.com	1792	172	3195	65721	0	110573	10	0	1
stackoverflow.com	1958	189	3159	73459	0	89317	8	0	1
www.amazon.com	3546	426	6907	290246	0	83180	57	0	4
www.amazon.in	9232	1275	24541	415482	0	419260	88	0	11
www.bbc.com	3901	492	7483	168880	0	132489	62	0	3
www.booking.com	10805	902	15151	521192	0	168810	32	0	4
www.buzzfeed.com	65584	10950	181999	3399266	161525	567398	21	1	4
www.mozilla.org	2117	283	3804	90004	0	149831	6	0	1
www.office.com	4145	476	7919	161807	0	93696	10	0	2
www.paypal.com	5286	405	9666	199482	0	75540	14	0	1
www.theguardian.com	3371	76	1095	174943	0	1145634	6	0	2
www.wikipedia.org	591	83	899	29769	0	104402	5	0	1
In-the-lab (Optimization Level 3)									
angularjs_require	11139	1300	65399	1028166	8356	3129	13	2	1
backbone	3517	246	10577	186378	8754	2666	12	2	1
backbone_require	9955	991	32405	462877	8356	2049	15	2	3
canjs	6225	454	23674	357109	8754	2917	8	2	1
canjs_require	12124	1154	45703	732586	8754	2689	37	2	1
dijon	4326	293	15176	199383	8754	2488	13	2	1
dojo	1798	729	4764	172332	8754	5939	27	2	3
enyo_backbone	13544	2472	17583	471740	8356	1295	29	2	1
exoskeleton	993	96	2604	63162	8754	2743	13	2	1
gwt	3814	5397	3911	529224	12069	902	8	3	1
jquery	4666	309	15895	201554	8391	2621	6	3	1
jsblocks	7505	344	35913	325880	8754	2843	6	2	1
knockback	6411	543	29659	367302	8754	3619	12	2	1
knockoutjs_require	1255	553	5294	128677	8356	2590	9	2	1
mithril	752	46	1713	36685	8754	1489	9	2	1
polymer	81	1299	166	195258	8523	105159	6	2	15
reagent	20485	1439	44242	1295032	8789	991	4	2	1
vanilla-es6	460	77	473	28407	8789	1761	10	2	1
vanillajs	646	101	2077	29879	8356	2613	10	2	2
vue	4582	289	11441	156097	8356	2592	7	2	1

5.2 Experimental Variables

In this section we will be describing the experimental variables involved in the experiment. Table 5.5 shows a summary of the variables.

Table 5.5: Experiment Variables

Variable Name	**Value**	**Type**
Energy Consumption	Joules (J)	Dependent
CPU Usage	Percentage (%)	
Memory consumption	KByte	
Loading Time	milliseconds (ms)	
FP	milliseconds (ms)	
FCP	milliseconds (ms)	
Packets Transferred	Total number of packets	
Bytes Transferred	Bytes	
JavaScript Code	Optimization level-0 Optimization level-1 Optimization level-2 Optimization level-3	Independent

Energy Consumption provides us with the battery's consumption of energy for loading the subjects of our experiment under the different treatments.

Performance metric is tracked by measuring different variables including CPU usage, memory consumption, loading time, first paint(FP), first contentful paint (FCP). *CPU usage* provides the percentage of CPU the mobile device uses to load the subjects. *Memory consumption* provides the amount of memory consumed by the mobile device to load the subjects. *Loading time* provides the time it takes the subject to fully load. *FP* provides the render time of any visual aspect different to what was on screen before navigation to the web page. *FCP* provides the render time of the first bit of content from the DOM.

Network usage metric is tracked by measuring the variables packets transferred and Bytes transferred. *Packets transferred* provides the total amount of packets transferred from the server to the device. *Bytes transferred* provides the total amount of Bytes transferred from the server to the device.

For the experiment, we also have an independent variable which is the JavaScript code. To detect JavaScript dead code, one of the analyzers combinations with the best f1score is

selected. The combination selected for this experiment consists of **dyn & tajs**. Furthermore, LacunaV2 have 4 different optimization levels as mentioned earlier which are the 4 treatments for our experiment.

OL0*Optimization level-0* does not perform any optimization, but provides an understanding about the analysis and possible detection of JavaScript dead code.

OL1*Optimization level-1* only replaces the function body but with a lazy loading mechanism which will retrieve the original function body when called.

OL2*Optimization level-2* removes the function body only without replacing it with other code. However, it keeps the function declaration to avoid having errors in the web page.

OL3*Optimization level-3* is the most aggressive strategy, it will remove all detected dead functions entirely to optimize the web application as much as possible making it an efficient strategy. However, this strategy is more inclined to have false positive errors.

5.3 Experimental Hypotheses

In this section we will define our null and alternative hypotheses for the results of the experiment to be able to answer the research questions of the paper.

The first null hypothesis states that the energy consumption used to load the treatments for all optimization levels are all equal. The alternative hypothesis states that the energy consumption is not equal for at least two optimization levels.

$$H_0^{energy} : \mu_{T-OL0} = \mu_{T-OL1} = \mu_{T-OL2} = \mu_{T-OL3}$$

$$H_a^{energy} : \mu_i \neq \mu_j; \exists i, j \in \{T-OL0, T-OL1, T-OL2, T-OL3\}$$

Where µ is equal to the mean of energy consumption and T-OL0, T-OL1, T-OL2, and T-OL3 are the treatments of different optimization levels.

The second null hypothesis states that the CPU usage used to load the treatments for all optimization levels are all equal. The alternative hypothesis states that the CPU usage is not equal for at least two optimization levels.

$$H_0^{cpu} : \lambda_{T-OL0} = \lambda_{T-OL1} = \lambda_{T-OL2} = \lambda_{T-OL3}$$

$$H_a^{cpu} : \lambda_i \neq \lambda_j; \exists i, j \in \{T-OL0, T-OL1, T-OL2, T-OL3\}$$

Where λ is equal to the mean of CPU usage and T-OL0 to T-OL3 are the treatments of different optimization levels.

The third null hypothesis states that the memory consumption used to load the treatments for all optimization levels are all equal. The alternative hypothesis states that the memory consumption is not equal for at least two optimization levels.

$$H_0^{mem} : \alpha_{T-OL0} = \alpha_{T-OL1} = \alpha_{T-OL2} = \alpha_{T-OL3}$$

$$H_a^{mem} : \alpha_i \neq \alpha_j; \exists i, j \in \{T-OL0, T-OL1, T-OL2, T-OL3\}$$

Where α is equal to the mean of memory consumption and T-OL0 to T-OL3 are the treatments of different optimization levels.

The fourth null hypothesis states that the loading time used to load the treatments for all optimization levels are all equal. The alternative hypothesis states that the loading time is not equal for at least two optimization levels.

$$H_0^{load} : \beta_{T-OL0} = \beta_{T-OL1} = \beta_{T-OL2} = \beta_{T-OL3}$$

$$H_a^{load} : \beta_i \neq \beta_j; \exists i, j \in \{T-OL0, T-OL1, T-OL2, T-OL3\}$$

Where β is equal to the mean of loading time and T-OL0 to T-OL3 are the treatments of different optimization levels.

The fifth null hypothesis states that the fp used to load the treatments for all optimization levels are all equal. The alternative hypothesis states that the fp is not equal for at least two optimization levels.

$$H_0^{fp} : \gamma_{T-OL0} = \gamma_{T-OL1} = \gamma_{T-OL2} = \gamma_{T-OL3}$$

$$H_a^{fp} : \gamma_i \neq \gamma_j; \exists i, j \in \{T-OL0, T-OL1, T-OL2, T-OL3\}$$

Where γ is equal to the mean of fp and T-OL0 to T-OL3 are the treatments of different optimization levels.

The sixth null hypothesis states that the fcp used to load the treatments for all optimization levels are all equal. The alternative hypothesis states that the fcp is not equal for at least two optimization levels.

$$H_0^{fcp} : \delta_{T-OL0} = \delta_{T-OL1} = \delta_{T-OL2} = \delta_{T-OL3}$$

$$H_a^{fcp} : \delta_i \neq \delta_j; \exists i, j \in \{T-OL0, T-OL1, T-OL2, T-OL3\}$$

Where δ is equal to the mean of fcp and T-OL0 to T-OL3 are the treatments of different optimization levels.

The seventh null hypothesis states that the packets transferred used to load the treatments for all optimization levels are all equal. The alternative hypothesis states that the packets transferred is not equal for at least two optimization levels.

$$H_0^{packet} : \zeta_{T-OL0} = \zeta_{T-OL1} = \zeta_{T-OL2} = \zeta_{T-OL3}$$

$$H_a^{packet} : \zeta_i \neq \zeta_j; \exists i, j \in \{T-OL0, T-OL1, T-OL2, T-OL3\}$$

Where ζ is equal to the mean of packets transferred and T-OL0 to T-OL3 are the treatments of different optimization levels.

The eighth null hypothesis states that the bytes transferred used to load the treatments for all optimization levels are all equal. The alternative hypothesis states that the bytes transferred is not equal for at least two optimization levels.

$$H_0^{byte} : \nu_{T-OL0} = \nu_{T-OL1} = \nu_{T-OL2} = \nu_{T-OL3}$$

$$H_a^{byte} : \nu_i \neq \nu_j; \exists i, j \in \{T-OL0, T-OL1, T-OL2, T-OL3\}$$

Where ν is equal to the mean of bytes transferred and T-OL0 to T-OL3 are the treatments of different optimization levels.

5.4 Experiment Design

In our experiment, the design encompasses two set of subjects for the in-the-lab(20 subjects) and real mobile web applications(16 subjects) experiments. However, both experiments are executed under the same exact environment. So, the overall experiment is comprised of a total of 36 subjects dealing with only one factor(JavaScript code) and 4 different treatments(OL0, OL1, OL2, and OL3). Therefore, our experiment design will follow the 1 factor many treatment strategy(1F-MT).

To ensure the collected measurements of the experiment are consistent, for each subject we will execute 20 runs. For each run, the various performance metrics, the two network usage metrics and the energy consumption metric are measured.

6

Experiment Execution

To describe the execution of the experiment, we will establish the specification and setup of the devices and tools used throughout the experiment.

6.1 Setup & Execution

Before we start the experiment we need to setup the devices and tools that we will use in the experiment. For the devices, we use one mobile phone and one machine to execute all runs of our experiment to have consistent results and reduce fluctuation in the results due to hardware changes. The specifications of the machine and the mobile device used in our experiment are as follows:

Samsung Galaxy J7 Duo

- Android version 8.0.0 Oreo
- 64 bit Octa Core Processor
- 139.5 mm (5.5") HD sAMOLED display
- 3 GB RAM
- 32 GB Memory

HP Laptop

- Ubuntu 19.10
- 64-bit OS type
- Intel Core i7-7500U CPU @ 2.70GHz x 4
- Intel HD Graphics 620 (Kaby Lake GT2) / AMD Radeon
- 1 TB Memory

An illustration of the experiment setup is shown in Figure 6.1. The devices used are connected together through a USB, this will allow the HP machine to communicate with Android phone.

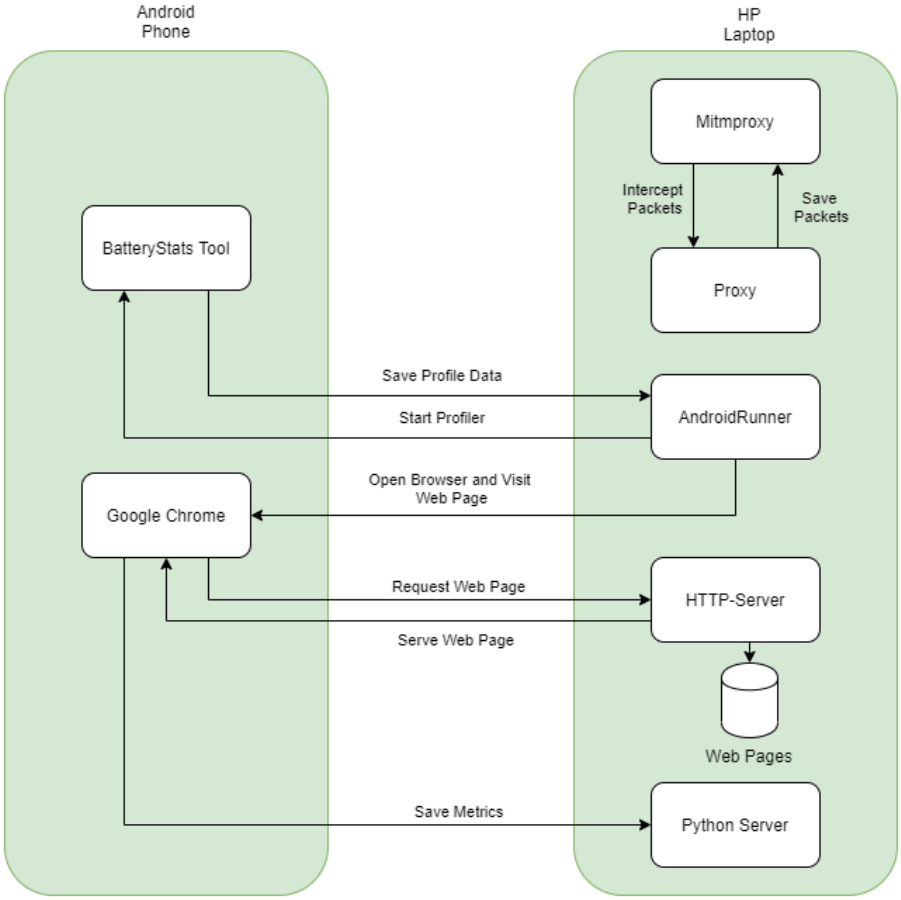


Figure 6.1: Experiment Setup Illustration

To access the subjects in the experiment, first we need to setup all subjects to be accessed through a HTTP-server. Both in-the-lab and real mobile web applications subjects,

including all of their modified subjects on which we applied the LacunaV2 tool to remove JavaScript dead code, are hosted on the HP machine.

Before we start the experiment, we need to make sure all metrics are ready to be collected. To collect the number packets and the bytes transferred for each subject we will have to setup a proxy on the android devices' internet connection. In the android device we manually set up a proxy with a proxy IP and proxy port. However, to be able to read the packets received by this proxy we run the mitmproxy tool on the HP machine. We execute mitmproxy through the command line as shown in Listing 6.1 which will start a server that will listen to the proxy and save the flows received to a text file.

Listing 6.1: Mitmproxy command

```
$ mitmproxy -p 8080 --set save_stream_file=flowDay27
```

To collect the Loading Time, FP, and FCP metrics we add a JavaScript code snippet shown in Figure 6.2 to the entrance html file of each subject. The code will measure and collect the metrics desired and sent to an executed python code which operates as a server hosted on the HP machine. Loading Time is calculated by subtracting the starting navigation time from the end loaded content event timing. Additionally, we use the perfumejs object to collect FP and FCP metrics. The perfumejs object collects several additional metrics such as navigationTiming, networkInfo and storageEstimate, however these metrics do not provide useful measurements for our experiment. Furthermore, the storageEstimate metric is collected, but returns zero and null values for its measurement since it calculates cache values which is cleared after every run by AndroidRunner. Finally, the server collects the request sent from the JavaScript code and writes each metric to a file in the HP machine.

Listing 6.2: JavaScript snippet for collecting metrics

```
<script src="/node_modules/perfume.js/dist/perfume.umd.min.js"></script>
<script>function xml_http_post(url, data, callback) {
var req = new XMLHttpRequest(); req.open("POST", url, true);
req.send(data);} setTimeout(function(){
if(LoadingEnd == 0){
LoadingEnd = window.performance.timing.domContentLoadedEventEnd;
LoadingTime = LoadingEnd - LoadingStart;}
objectToSend="{'WebName':"+window.location.href+",
'Load':"+LoadingTime+",␣'Start':"+LoadingStart+",
'End':"+LoadingEnd+",
```

```
'PerfumeResult':"+JSON.stringify(perfumeResults)+"}";
xml_http_post("http://192.168.178.178:8081/",objectToSend,null);
}, 10000); perfumeResults = [];
LoadingTime =
(window.performance.timing.domContentLoadedEventEnd-
window.performance.timing.navigationStart);
LoadingStart = window.performance.timing.navigationStart;
LoadingEnd = window.performance.timing.domContentLoadedEventEnd;
const perfume = new Perfume({  analyticsTracker: (options) => {
const { metricName, data, eventProperties, navigatorInformation } = options;
perfumeResults.push(options); } });
function load_log() {LoadingTime =
(window.performance.timing.domContentLoadedEventEnd-
window.performance.timing.navigationStart);
LoadingStart = window.performance.timing.navigationStart;
LoadingEnd= window.performance.timing.domContentLoadedEventEnd};
window.addEventListener ?window.addEventListener("load",load_log, true) :
window.attachEvent && window.attachEvent("onload", load_log);</script>
```

After the HTTP-server, we setup the mitmproxy and the server to collect metrics. then we start the execution of AndroidRunner[1] which executes the experiment using a configuration file. This file is structured to define the details of the experiment, it illustrates the set of web pages that it will execute, the amount of runs for each web page, as well as the duration of each run of the experiment. After starting AndroidRunner, the experiment is executed on the android device connected with machine. For each run, Google Chrome(Chrome 81.0.4044.138) is launched and the web page is requested from the machine using the IP address of the HTTP-server and loaded to the browser of the android phone. During the duration of loading the web pages, mitmproxy collects and writes the packets transferred between the android device and the machine. After the web page is loaded, the JavaScript snippet mentioned earlier is executed and loading time, FP, and FCP are collected and written to a file on the machine. After each run, a file containing battery statistics such as energy consumption, memory consumption and CPU usage is collected and saved on the HP machine. Furthermore, after the file is saved, the tool resets the android device to its initial state, therefore every run is executed with the android device is in its initial state and they are not influenced differently. This cool down procedure takes a duration of exactly 2 minutes.

Each subject run take up to 30 seconds to execute and loading the web page. The experiment executes 20 runs for each variant of each web application, we have 36 subjects

[1]https://github.com/S2-group/android-runner

selected (16 real mobile web applications and 20 in-the-lab) and including the cool down time of 2 minutes. Therefore, the total time to complete the entire experiment is 7200 minutes.

7

Results

The results will comprise of some plots to illustrate the metrics collected. Furthermore, statistical analysis tests are executed on the data of all metrics collected, which will assist to answer the research questions of the project.

7.1 Result Representation

For the results of each metric, we present some plots to illustrate the possible differences in the data collected between the four variation of each subject.

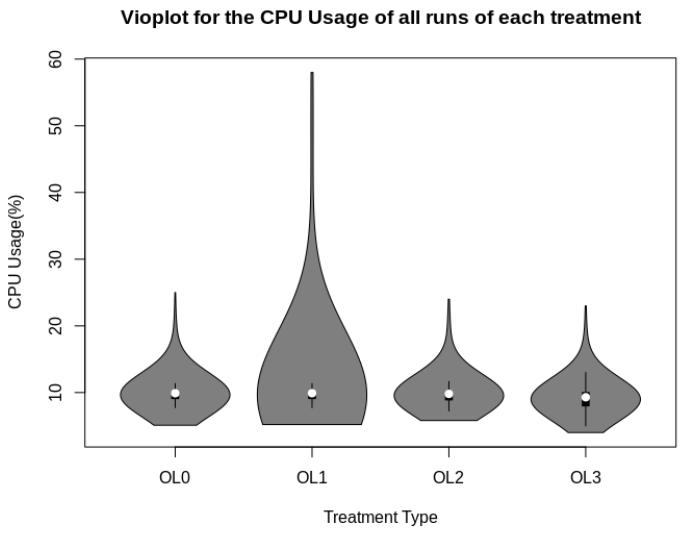


Figure 7.1: Violin plot of CPU usage across the treatments

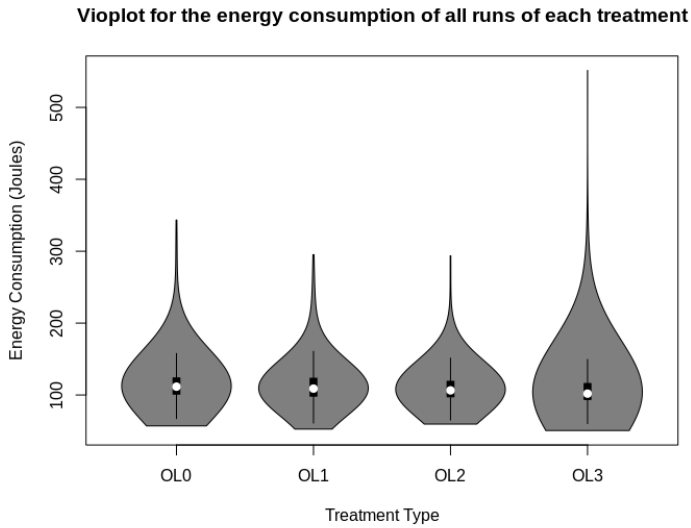


Figure 7.2: Violin plot of energy consumption across the treatments

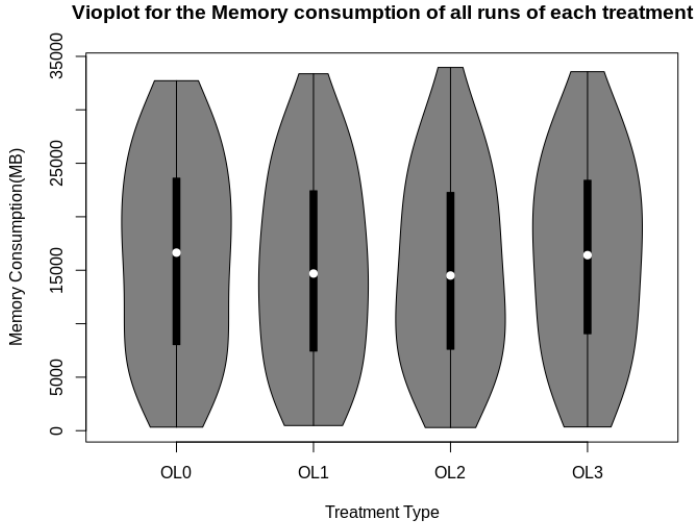


Figure 7.3: Violin plot of memory consumption across the treatments

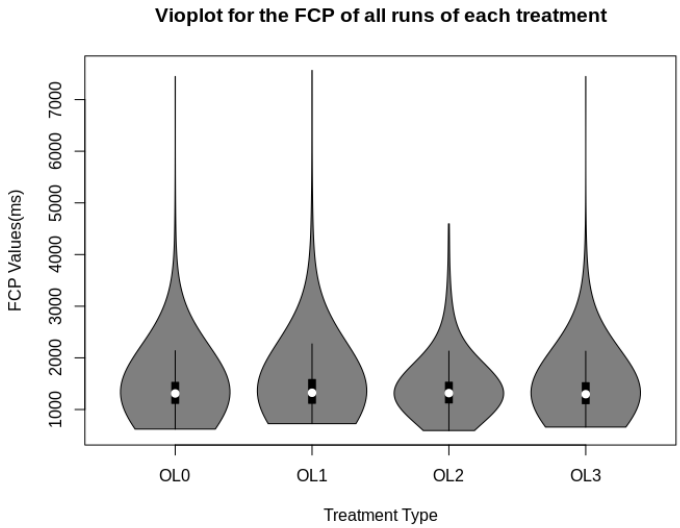


Figure 7.4: Violin plot of FCP across the treatments

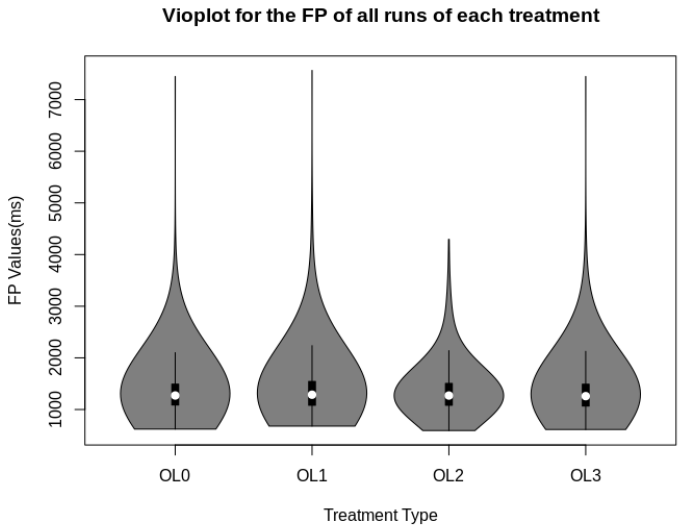


Figure 7.5: Violin plot of FP across the treatments

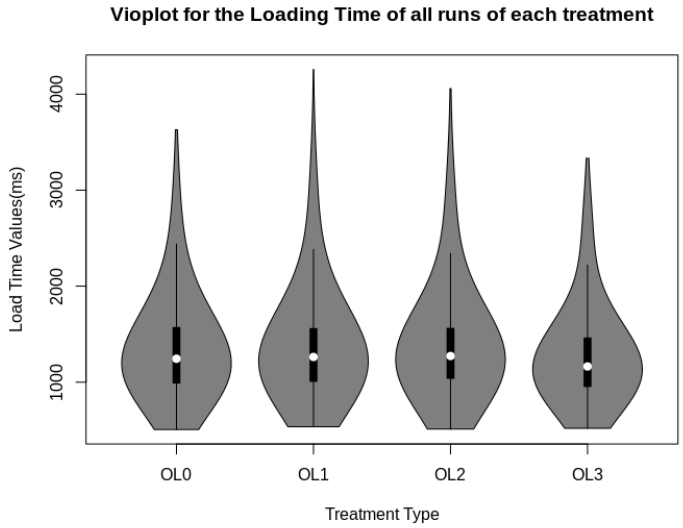


Figure 7.6: Violin plot of loading times across the treatments

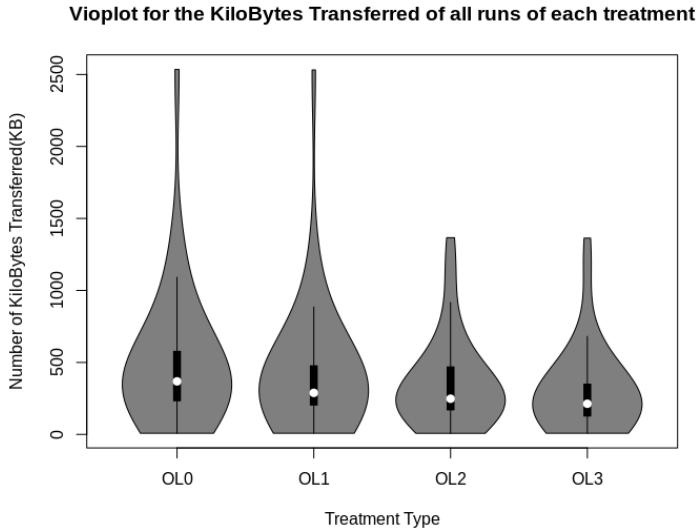


Figure 7.7: Violin plot of kilobytes transferred across the treatments

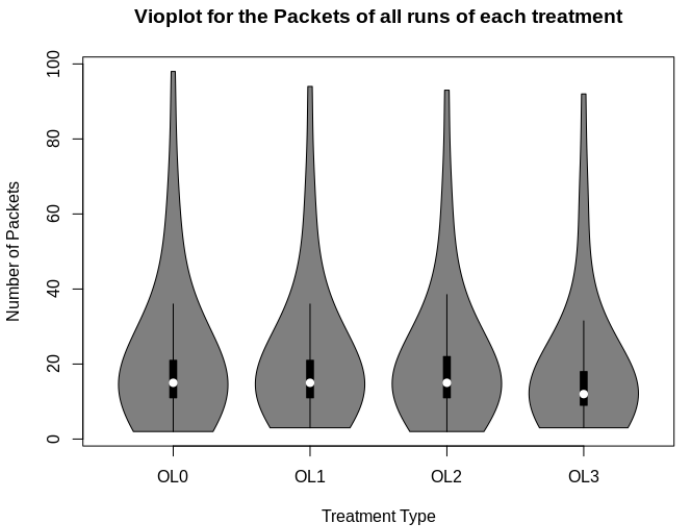


Figure 7.8: Violin plots of packets transferred across the treatments

In Figures 7.1 to 7.8, we present all the results of each metric over all the four treatments OL0, OL1, OL2, and OL3 in violin plots. We observe that predominantly in the plots the values are slightly decreasing across the treatment advancing from OL0 to OL3. However, some plots such as FCP, FP, and memory consumption the difference across the treatments are not large. Furthermore, Figure 7.7 illustrates the plot of the Byte transferred metric in Kilobytes. The reason for this conversion is to avoid the numbers in y-axis overlapping each other and provide a more coherent plot.

Some plots show a spike in values across some of the treatments such CPU usage and energy consumption. These spikes could occur for multiple reasons such as the unusual spike in use for a built-in service of the android device which increases the CPU usage of the device. Furthermore, the browser waiting longer to receive response from the HTTP-server in the hp machine which causes the browser to consume more energy and CPU from the device. To further provide explanation of the spikes and the distribution of the results of each metric, we provide Tables 7.1 to 7.8 which present *Min*, *Max*, *Average*, *Median*, *Standard Deviation* and *Coefficient of Variation* values to overview the data range of all four treatments of each metric. The Min, Max and Median presents the minimum, maximum and middle points of a range of values respectively. The average provides the mean value of the range of values. The standard deviation is a value indicating by how much a range of values differ from the mean of the range. The coefficient of variation is a value indicating the ratio of the standard deviation to the mean of a range of values(11).

Table 7.1: Overview of CPU usage values

	Min	Max	Average	Median	Standard Deviation	Coefficient of Variation
OL0	5.1	25	9.888889	9.9	1.873492	0.1894543
OL1	5.2	58	10.935	9.9	6.790073	0.6209486
OL2	5.8	24	9.796667	9.8	2.020435	0.206237
OL3	4	23	9.242083	9.3	2.042904	0.2210436

Table 7.2: Overview of energy consumption values

	Min	Max	Average	Median	Standard Deviation	Coefficient of Variation
OL0	57.00978	343.3014	114.8699	111.5859	26.69559	0.2323984
OL1	52.50492	295.146	114.1651	108.8555	27.84051	0.2438618
OL2	59.49522	293.5926	109.3339	106.4079	21.38611	0.1956037
OL3	50.4855	551.457	107.0477	101.8912	30.22908	0.2823889

Table 7.3: Overview of Memory consumption values

	Min	Max	Average	Median	Standard Deviation	Coefficient of Variation
OL0	329.3936	32722.3	16109.54	16643.56	9018.913	0.5598491
OL1	477.876	33373.42	15129.37	14691.86	8811.419	0.5824047
OL2	287.5293	33966.32	15181.16	14506.16	8700.293	0.5730979
OL3	348.3135	33571.09	16238.39	16413.71	8824.985	0.5434644

Table 7.4: Overview of FCP values

	Min	Max	Average	Median	Standard Deviation	Coefficient of Variation
OL0	621.3	7447.5	1390.74	1310.70	472.1034	0.339462
OL1	726.2	7564.8	1449.572	1324.25	583.096	0.4022538
OL2	592.2	4594.4	1425.541	1317.65	486.1704	0.3410427
OL3	660.3	7447.5	1399.274	1296.80	485.6885	0.3471004

Table 7.5: Overview of FP values

	Min	Max	Average	Median	Standard Deviation	Coefficient of Variation
OL0	621.3	7447.5	1352.555	1271.65	442.4467	0.3271192
OL1	679.1	7564.8	1396.839	1286.60	530.1797	0.3795567
OL2	592.2	4294	1373.022	1270.50	460.399	0.3353182
OL3	612.7	7447.5	1357.42	1259.05	483.1468	0.3559302

Table 7.6: Overview of Loading Time values

	Min	Max	Average	Median	Standard Deviation	Coefficient of Variation
OL0	506	3630	1358.165	1244.0	507.6377	0.3737672
OL1	535	4258	1376.05	1262.5	531.5773	0.3863067
OL2	511	4058	1394.511	1272.5	542.0021	0.3886682
OL3	519	3332	1286.917	1163.5	486.0178	0.3776606

Table 7.7: Overview of Bytes transferred values (in KB)

	Min	Max	Average	Median	Standard Deviation	Coefficient of Variation
OL0	8.107422	2535.027	497.0852	368.4629	463.0651	0.9315608
OL1	8.107422	2531.227	415.3282	288.5928	425.1328	1.023607
OL2	7.347656	1366.257	353.8287	247.8379	305.0427	0.8621198
OL3	8.194336	1363.725	293.6379	212.9922	278.0986	0.9470801

Table 7.8: Overview of packets transferred values

	Min	Max	Average	Median	Standard Deviation	Coefficient of Variation
OL0	2	98	21.68333	15	18.63483	0.8594082
OL1	3	94	21.68889	15	18.31658	0.8445143
OL2	2	93	21.87778	15	18.26102	0.8346835
OL3	3	92	19.10833	12	18.31541	0.9585037

In Table 7.1 we include the values to overview the CPU usage data for all variants. For Max, Average, Standard Deviation and Coefficient of variation values, we observe that the values in OL2 and OL3 decrease compared to OL0, however the value increases in OL1. For the Median values, we observe that that OL0 and OL1 are similar values and then the value decrease in OL2 and OL3 compared to OL0. Finally for the Min values, we observe that the value in OL1 and OL2 increase compared in OL2, however the value decreases in OL3.

In Table 7.2 we include the values to overview the energy consumption data for all variants. For the Standard Deviation and Coefficient of Variation values, we observe the values increasing from OL0 to OL1 and OL3, however decreasing to OL2. For the Median and Average values, the decrease occurs in all treatments(OL1, OL2, OL3) compared to OL0. For the Max values, we observe a decrease in OL1 and OL2 and an increase in OL3 compared to OL0. Finally for the Min values, we observe a decrease in OL1 and OL3 and an increase in OL2 compared to OL0.

In Table 7.3 we include the values to overview the memory consumption data for all variants. For the Standard Deviation and Median values, we observe the values decrease from OL0 to OL1, OL2 and OL3. For the Average values, the decrease occurs in OL1, OL2 compared to OL0, however an increase in OL3. For the Min values, we observe a decrease only in OL2 and an increase in OL1 and OL3 compared to OL0. For the Coefficient of Variation values, we observe a decrease only in OL3 and an increase in OL1 and OL2. Finally for Max values, we observe all treatments to increase compared to OL0.

In Table 7.4 we include the values to overview the FCP data for all variants. For the Standard Deviation, Coefficient of Variation and Average values, we observe all treatments to increase compared to OL0. For the Median values, a decrease occurs only in OL3 and an increase occurs in OL1 and OL2 compared to OL0. For the Min values, we observe a decrease only in OL2 and an increase in OL1 and OL3 compared to OL0. Finally for the Max values, we observe a OL3 to have an equal value compared to OL0, however we observe a decrease in OL2 and an increase in OL1.

In Table 7.5 we include the values to overview the FP data for all variants. For the Standard Deviation, Coefficient of Variation, Median, and Average values, we observe all treatments to increase compared to OL0. For the Min values, we observe a decrease in OL2 and OL3 and an increase in OL1 compared to OL0. Finally for the Max values, we observe a OL3 to have an equal value compared to OL0, however we observe a decrease in OL2 and an increase in OL1.

In Table 7.6 we include the values to overview the loading Time data for all variants. For the Max, Standard Deviation, Median, and Average values, we observe a decrease only in OL3 and an increase in OL1 and OL2 compared to OL0. Finally for the Min and Coefficient of Variation values, we observe all treatments to increase compared to OL0.

In Table 7.7 we include the values to overview the bytes transferred data for all variants. For the Max, Standard Deviation, Median, and Average values, we observe all treatments to decrease compared to OL0. For the Min values, we observe a OL1 to have an equal value compared to OL0, however we observe a decrease in OL2 and an increase in OL3. Finally for the Coefficient of Variation values, we observe a decrease only in OL2 and an increase in OL1 and OL3 compared to OL0.

In Table 7.8 we include the values to overview the packets transferred data for all variants. For the Standard Deviation and Max values, we observe all treatments to decrease compared to OL0. For the Min values, we observe a OL2 to have an equal value compared to OL0, however an increase in OL1 and OL3. For the Average values, we observe a decrease only in OL3 and an increase in OL1 and OL2 compared to OL0. For the Median values, we observe OL1 and OL2 to have an equal value compared to OL0 however a decrease in OL3. Finally for the Coefficient of Variation values, we observe a decrease in OL1 and OL2 and an increase in OL3 compared to OL0.

Looking at the *Average* value presented within the table we are able to detect a possible decrease of the variables across certain treatments. However, we can not make a conclusion from only the violin plots and the tables. We need to perform statistical tests on the results obtained to verify the decrease we observe in some of the data**??**.

7.2 Normality Check

To be able to select the statistical test to perform, we check the normality distribution assumption of the data for each metric. To visually check the normality of the distribution, we generated Q-Q plots for each variant of each metric against a random sample taken from a normal distribution (see Figure 12.1 to Figure 12.32). The Q-Q plots show some data distant from the normal line created, therefore verifying that the data of all variants about all metrics are not normally distributed.

To get statistical indication for the normality of the distribution, we perform Shapiro-Wilk test. The result of applying this test is shown in Table 7.9. The table shows all p-values to be below 0.05, thus rejecting the null hypothesis that the data lies within a

normal distribution. Therefore, we can conclude the data of all metrics we measured are not normally distributed.

Table 7.9: Shapiro-Wilk normality test results

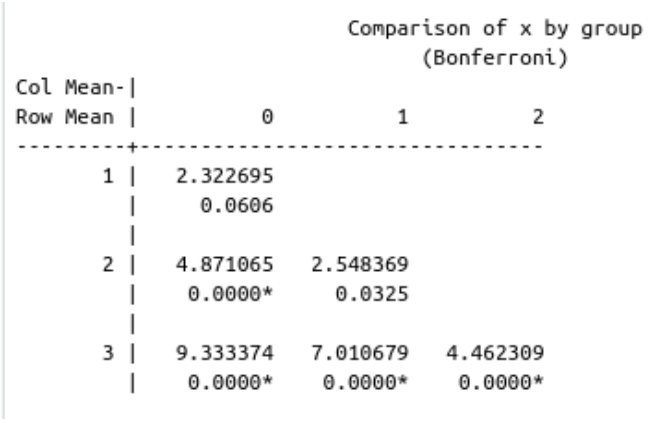
Treatments	Variables	W	p-value
OL0	Energy consumption	0.73158	< 2.2e-16
	CPU usage	0.66946	< 2.2e-16
	Memory consumption	0.95604	7.59e-14
	FCP	0.6981	< 2.2e-16
	FP	0.72664	< 2.2e-16
	Loading time	0.87192	< 2.2e-16
	Bytes transferred	0.73883	< 2.2e-16
	Packets transferred	0.69907	< 2.2e-16
OL1	Energy consumption	0.77307	< 2.2e-16
	CPU usage	0.31612	< 2.2e-16
	Memory consumption	0.95967	3.61e-13
	FCP	0.6653	< 2.2e-16
	FP	0.71351	< 2.2e-16
	Loading time	0.84494	< 2.2e-16
	Bytes transferred	0.66056	< 2.2e-16
	Packets transferred	0.70636	< 2.2e-16
OL2	Energy consumption	0.81326	< 2.2e-16
	CPU usage	0.68922	< 2.2e-16
	Memory consumption	0.96377	2.373e-12
	FCP	0.78959	< 2.2e-16
	FP	0.81121	< 2.2e-16
	Loading time	0.85691	< 2.2e-16
	Bytes transferred	0.80763	< 2.2e-16
	Packets transferred	0.70877	< 2.2e-16
OL3	Energy consumption	0.58649	< 2.2e-16
	CPU usage	0.80049	< 2.2e-16
	Memory consumption	0.96595	6.823e-12
	FCP	0.72163	< 2.2e-16
	FP	0.72672	< 2.2e-16
	Loading time	0.86229	< 2.2e-16
	Bytes transferred	0.74328	< 2.2e-16
	Packets transferred	0.64392	< 2.2e-16

7.3 Hypothesis Testing

In section 7.1, most metrics displayed a decrease in values in certain treatments in the violin plots and the overview tables. However, this decrease might not be significant to perform the JavaScript dead code removal process. Furthermore in section 7.2, the result of the normality check showed that the data of the variables are not normally distributed. Therefore, we select to perform the *Krustal-Wallis test*[1]. This test will allow us to test the null hypothesis we determined earlier that the measurement used of each metric across all treatments of our experiment are equal without the need for the data obtained to be normally distributed. Furthermore, to detect which variants are significantly different compared to the original variant(OL0), we perform a *Dunn's Test*(12) with "Bonferroni" correction. This test performs a pairwise comparison to provide a p-value for each pair of the variants, which will help us verify the impact of JavaScript dead code on the variants compared to the original variant.

The first metric we test its null hypothesis is *energy consumption*. The application of Krustal-Wallis test on the energy consumption data produces a p-value of of less than 2.2e-16 which is less than the 5% significance level, therefore allowing us to reject the null hypothesis stating that the energy consumption measures across the treatments are equal. The application of Dunn's test produces the p-values shown in the first column(bottom values) of Figure 7.9. The resulting p-value for the OL0-OL1 pair is "0.0606", whereas both the OL0-OL2 and OL0-OL3 pairs obtained p-values of "0.0000*" which is two different p-values close to zero. The results show clear significance difference between OL0 compared to both OL2 and OL3 only, therefore verifying that the presence of JavaScript dead code in web applications consumes more energy at run-time.

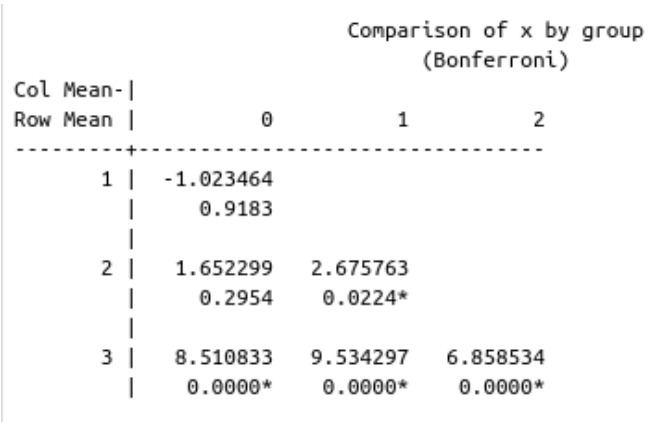
[1] https://statistics.laerd.com/spss-tutorials/kruskal-wallis-h-test-using-spss-statistics.php



```
                        Comparison of x by group
                                (Bonferroni)
Col Mean-|
Row Mean |          0          1          2
---------+---------------------------------
       1 |   2.322695
         |     0.0606
         |
       2 |   4.871065   2.548369
         |    0.0000*     0.0325
         |
       3 |   9.333374   7.010679   4.462309
         |    0.0000*    0.0000*    0.0000*
```

Figure 7.9: Dunn test for energy consumption

The second metric we test its null hypothesis is *CPU usage*. The application of Krustal-Wallis test on the CPU usage data produces a p-value of less than 2.2e-16, therefore allowing us to reject the null hypothesis stating that the CPU usage measures across the treatments are equal. Furthermore, the application of Dunn's test produces the p-values shown in the first column(bottom values) of Figure 7.10. The resulting p-value for the OL0-OL1 pair is "0.9183", whereas the OL0-OL2 and OL0-OL3 pairs obtained p-values of "0.2954" and "0.0000*" respectively. The results show clear significance difference between the OL0-OL3 pair, therefore verifying that the presence of JavaScript dead code in web applications uses more CPU at run-time.



```
                        Comparison of x by group
                                (Bonferroni)
Col Mean-|
Row Mean |          0          1          2
---------+---------------------------------
       1 |  -1.023464
         |     0.9183
         |
       2 |   1.652299   2.675763
         |     0.2954    0.0224*
         |
       3 |   8.510833   9.534297   6.858534
         |    0.0000*    0.0000*    0.0000*
```

Figure 7.10: Dunn test for CPU usage

The third metric we test its null hypothesis is *Memory consumption*. The application of Krustal-Wallis test on the memory consumption data produces a p-value of 0.02314, therefore we reject the null hypothesis stating that the memory consumption measures

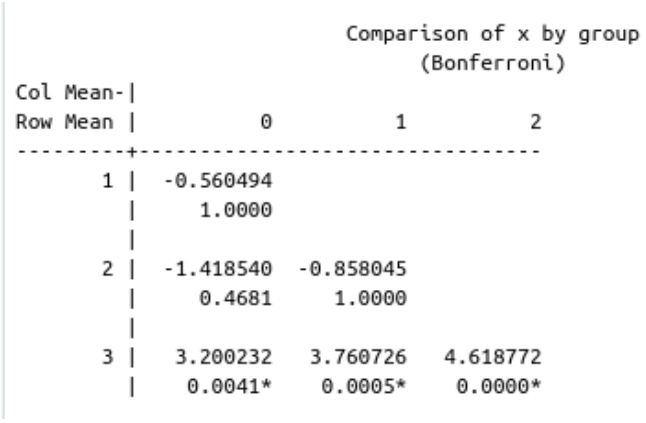
across the treatments are equal. Furthermore, Dunn's test produces the p-values shown in the first column(bottom values) of Figure 7.11. The resulting p-values of the OL0-OL1, OL0-OL2 and OL0-OL3 are "0.1170", "0.1442" and "1.0000" respectively. The result does not show a significance difference between OL0 and the other treatments, therefore we can not verify that the presence of JavaScript dead code in web applications consumes more memory at run-time. The reason for the Krustal-Wallis test showing that the memory consumption across the treatments are not equal might be because of a possible difference between OL1-OL3 as well as between OL2-OL3. Dunn's test provides p-value of "0.0543" for the pair OL1-OL3 and "0.0684" for the pair OL2-OL3 which are really close to the 5% significance level.

Comparison of x by group
(Bonferroni)

Col Mean- / Row Mean	0	1	2
1	2.064188		
	0.1170		
2	1.976846	-0.087341	
	0.1442	1.0000	
3	-0.299673	-2.363861	-2.276520
	1.0000	0.0543	0.0684

Figure 7.11: Dunn test for memory consumption

The fourth metric we test its null hypothesis is *Loading Time*. The application of Krustal-Wallis test on the loading Time data produces a p-value of 2.071e-05, therefore we reject the null hypothesis stating that the loading time measures across the treatments are equal. Additionally, Dunn's test produces the p-values shown in the first column(bottom values) of Figure 7.12. The resulting p-values of the OL0-OL1, OL0-OL2 and OL0-OL3 pairs are "1.0000", "0.4681" and "0.0041*" respectively. The result shows a significant difference between OL0-OL3, therefore verifying that the presence of JavaScript dead code in web applications have longer loading times at run-time.



Comparison of x by group (Bonferroni)			
Col Mean / Row Mean	0	1	2
1	-0.560494		
	1.0000		
2	-1.418540	-0.858045	
	0.4681	1.0000	
3	3.200232	3.760726	4.618772
	0.0041*	0.0005*	0.0000*

Figure 7.12: Dunn test for loading time

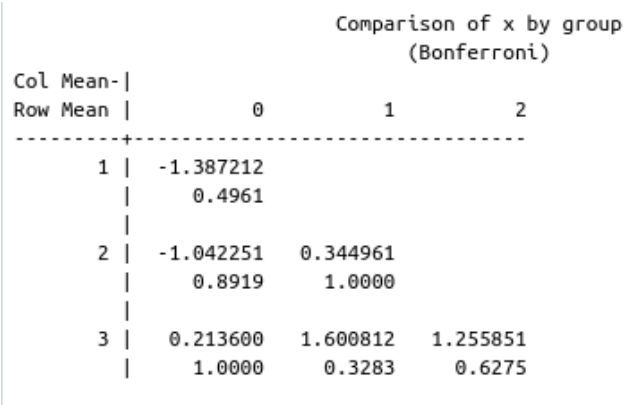
The fifth metric we test its null hypothesis is *FP*. The application of Krustal-Wallis test on the FP data produces a p-value of 0.4925, therefore we can not reject the null hypothesis stating that the FP measures across the treatments are equal. Furthermore, the Dunn's test produces the p-values shown in the first column(bottom values) of Figure 7.13. The resulting p-values of OL0-OL1, OL0-OL2 and OL0-OL3 pairs are "1.0000" for all the three pairs. The result shows no significant difference between OL0 and the other treatments, therefore verifying that the presence of JavaScript dead code in web applications does not effect the FP measures at run-time.

Comparison of x by group (Bonferroni)			
Col Mean / Row Mean	0	1	2
1	-0.946733		
	1.0000		
2	-0.245228	0.701505	
	1.0000	1.0000	
3	0.582599	1.529332	0.827827
	1.0000	0.3785	1.0000

Figure 7.13: Dunn test for FP

The sixth metric we test its null hypothesis is *FCP*. The application of Krustal-Wallis test on the FCP data produces a p-value of 0.3009, therefore we can not reject the null hypothesis stating that the FCP measures across the treatments are equal. Furthermore,

the Dunn's test produces the p-values shown in the first column(bottom values) of Figure 7.14. The resulting p-values of OL0-OL1, OL0-OL2 and OL0-OL3 pairs are "0.4961", "0.8919" and "1.0000" respectively. The result shows no significant difference between OL0 and the other treatments, therefore verifying that the presence of JavaScript dead code in web applications does not effect the FCP measures at run-time.

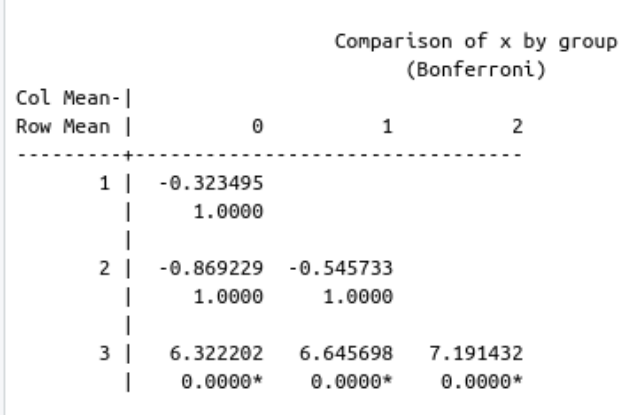


Comparison of x by group
(Bonferroni)

Col Mean / Row Mean	0	1	2
1	-1.387212		
	0.4961		
2	-1.042251	0.344961	
	0.8919	1.0000	
3	0.213600	1.600812	1.255851
	1.0000	0.3283	0.6275

Figure 7.14: Dunn test for FCP

The seventh metric we test its null hypothesis is *Packets Transferred*. The application of Krustal-Wallis test on the packets transferred data produces a p-value of 8.919e-15, therefore allowing us to reject the null hypothesis stating that the packets transferred measures across the treatments are equal. Additionally, Dunn's test produces the p-values shown in the first column(bottom values) of Figure 7.15. The resulting p-values of OL0-OL1, OL0-OL2 and OL0-OL3 pairs are "1.0000", "1.0000" and "0.0000*" respectively. The result shows a significant difference between OL0-OL3, therefore verifying that the presence of JavaScript dead code in web applications includes higher packets transferred at run-time.

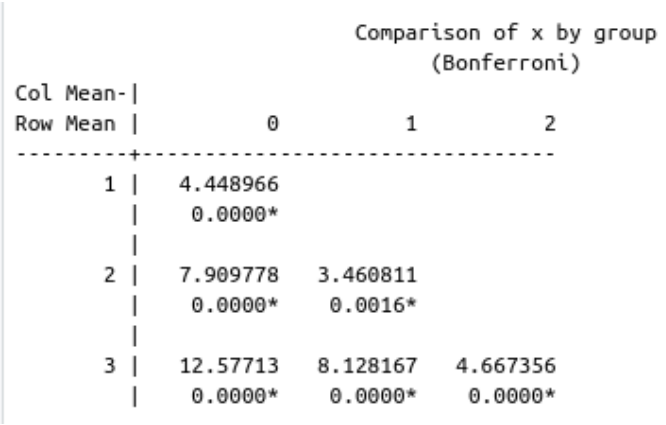


Comparison of x by group
(Bonferroni)

Col Mean- Row Mean	0	1	2
1	-0.323495 1.0000		
2	-0.869229 1.0000	-0.545733 1.0000	
3	6.322202 0.0000*	6.645698 0.0000*	7.191432 0.0000*

Figure 7.15: Dunn test for packets transferred

The final metric we test its null hypothesis is *Bytes Transferred*. The application of Krustal-Wallis test on the bytes transferred data produces a p-value of less than 2.2e-16, therefore allowing us to reject the null hypothesis stating that the Bytes transferred measures across the treatments are equal. Furthermore, Dunn's test produces p-values shown in the first column(bottom values) of Figure 7.16. The resulting p-values of OL0-OL1, OL0-OL2 and OL0-OL3 pairs are "0.0000*" for all the three pairs. The result shows a significance difference between all the three pairs, therefore verifying that the presence of JavaScript dead code in web application includes higher bytes transferred at run-time.



Comparison of x by group
(Bonferroni)

Col Mean- Row Mean	0	1	2
1	4.448966 0.0000*		
2	7.909778 0.0000*	3.460811 0.0016*	
3	12.57713 0.0000*	8.128167 0.0000*	4.667356 0.0000*

Figure 7.16: Dunn test for bytes transferred

8

Discussion

The outcome of our experiment provides a diverse analysis of dead JavaScript removal which rest on the metrics elected. To further understand outcome of JavaScript dead code removal, we need to answer the **RQs** of our paper.

To answer ***RQ1*** we observe the average value in the overview table, the result of testing the null hypothesis and the result of Dunn's test for the energy consumption variable. The result of Krustal-Wallis test shows that we reject the null hypothesis. Therefore, we can say that there is a statistically significant difference in energy consumption of web applications across the optimization levels treatments.

The overview table for energy consumption shows a decrease in the average for OL1, OL2 and OL3 compared to the OL0 variant. However, Dunn's test shows that there is a significant different only between OL0-OL2 and between OL0-OL3, thus, the decrease is significant in only these two pairs. The presence of JavaScript dead code in the subjects differ in terms of average energy consumption by "5.536" and "7.8222" Joules when compared to the same subjects with OL2 and OL3 approaches respectively. Therefore, we conclude JavaScript dead code negatively impacts the energy consumption of mobile web applications.

The obtained result of the energy consumption metric allows the developers to understand that while developing a web application they should only use JavaScript functions that are necessary for the web application to perform its functionality as well as try to import only the JavaScript functions that are needed from a JavaScript framework/library. Otherwise, these additional functions(JavaScript dead code) in the web application will not be executed and will cause the web application to consume more energy. Additionally, based on our experiment results, developers can achieve better energy consumption when removing JavaScript dead code using OL2 or OL3 method.

To answer ***RQ2*** we observe the average value in the overview tables, the results of the hypothesis testing and the results of Dunn's test for all metrics belonging to the performance metric. Three from the five metrics belonging to the performance metric provided similar conclusion for testing the null hypothesis. The three metrics are loading time, memory consumption and CPU usage. The results of Krustal-Wallis test for these three metrics show that we reject the null hypothesis for each respective metric. Therefore, we can say that there is a statistically significant difference in loading time, memory consumption and CPU usage of web applications across the optimization levels variants.

The overview tables show different transitions in the average across the variants of the three metrics. The overview table of CPU usage shows a decrease in the average for OL2 and OL3, however an increase for OL1 compared to OL0. The overview table of loading time shows a decrease in the average only for OL3, however an increase for OL1 and OL2 compared to OL0. Finally, the overview table of memory consumption shows a decrease in the average for OL1 and OL2, however an increase for OL3 compared to OL0. The application of Dunn's test shows that for CPU usage and loading time only the difference between OL0-OL3 is significance. The difference between OL0-OL3 is a decrease from OL0 to OL3 by "0.646806" percent(%) and "71.248" milliseconds(ms) for CPU usage and loading time respectively. Additionally, Dunn's test shows that for memory consumption the differences of the average value we observe in the overview table are not significant. Therefore, we can deduce that JavaScript dead code negatively impacts CPU usage and loading time, however has no effect on memory consumption.

The other two metrics belonging to the performance metrics are FP and FCP. These two metrics provided similar results to each other in terms of testing the null hypothesis, however different to the other five metrics. The result of testing the hypothesis for these two metrics shows that we can not reject the null hypothesis. Therefore, we can say that there is no statistically significant difference in FP and FCP of web applications across the optimization level treatments.

The overview tables for both FP and FCP show an increase in the average for OL1, OL2 and OL3 compared to the OL0 variant. However, testing the null hypothesis showed that there is no significant difference between the treatments. Furthermore, the application of Dunn's test verifies that there is not significant difference between OL1, OL2 and OL3 when compared to OL0. Therefore, we deduce that JavaScript dead code has no impact on the FP and FCP measures in mobile web applications. Finally, we can conclude that JavaScript dead code negatively impact some performance metrics of mobile web applications.

The obtained results of the variables belonging to the performance metric allows the developers to understand that while developing a web application they should only use JavaScript functions that are necessary for the web application to perform its functionality and try to import only the JavaScript functions that are needed from a JavaScript framework/library. Otherwise, these additional functions(JavaScript dead code) in the web application will not be executed and will cause the web application to use more CPU and have have higher loading time. Furthermore, based on our experiment results, developers can achieve better CPU usage and loading time when removing JavaScript dead code using OL3 method.

To answer ***RQ3*** we observe the average value in the overview tables, the results of the hypothesis testing and the results of Dunn's test for both metrics belonging to the network usage metric. The two metrics belonging to the network usage metric are packets transferred and bytes transferred. The application of Krustal-Wallis test shows that we reject both null hypothesis for packets and bytes transferred. Therefore, we can say that there is a statistically significant difference in packets transferred and bytes transferred of web applications across the optimization levels treatments.

The overview table of packets transferred shows a decrease in the average only for OL3, however an increase for OL1 and OL2 compared to OL0. Additionally, the overview table of bytes transferred shows a decrease in the average for OL1, OL2 and OL3 compared to OL0. The application of Dunn's test shows that for packets transferred the significance difference is between OL0-OL3. The difference between the pair OL0-OL3 is a decrease from OL0 to OL3 by "2.575" packets for packets transferred. Furthermore, for bytes transferred the outcome of Dunn's test shows that there is a significant difference between OL0-OL1, OL0-OL2 and OL0-OL3. For the three pair OL0-OL1, OL0-OL2 and OL0-OL3 the difference is a decrease from OL0 to each variant of each pair by "81.757", "143.2565" and "497.0852" KiloBytes(KB) respectively. Therefore, we deduce that JavaScript dead code has a negative impact on both network usage metrics of mobile web applications.

The obtained results of the variables belonging to the network usage metric allows the developers to understand that while developing a web application they should only use JavaScript functions that are necessary for the web application and try to import only the JavaScript functions that are needed from a JavaScript framework/library. Otherwise, these JavaScript dead code in the web application will not be executed and will cause the web application to include higher packets and bytes transferred throughout the network. Furthermore, based on our experiment results, developers can achieve better packets transferred when removing JavaScript dead code using OL3 method. Additionally, they

can achieve better bytes transferred when removing JavaScript dead code using OL1, OL2 or OL3.

9

Threats To Validity

9.1 External Validity

This validity addresses the generalisability of the experiment results. The results should be valid for the entire population targets the subjects are selected from. The two sets of subjects web applications of the experiment are selected from Tranco List and TodoMVC collection respectively. From which, 16 subjects from Tranco list and 20 subjects from TodoMVC collection are selected based on inclusion criteria. Since LacunaV2 is unstable, the inclusion criteria drops large portion of web apps from the population target. Thus, the inclusion criteria makes it difficult to generalise the results.

Additionally, the experiment is performed using devices with relatively good specification(e.g. Samsung Galaxy J7 Duo and HP "1TB Memory" Laptop). Performing all runs of the experiment under these tools we can rationally certain that the experiment setting can reflect similarly to a real world scenario. However, different and newer devices and machines may generate to different measurements of the metrics collected. Therefore, additional replications of the experiment will be beneficial to support the results of the experiment and reduce the threat to validity.

9.2 Internal Validity

The maturation threat addresses the possibility of the subjects reacting differently between trials over a period of time. However, this threat is mitigated by a two minutes cool down between runs. This cool down allows buffers or caches that might be used by the operating system of the mobile device or the machine to timeout. Additionally, the cache of the

browser used to load the subjects is cleared and reset after each run to mitigate its effect on the measurements.

Furthermore, the reliability of the measures can be affected by several factors such as services consuming assets of the device, and brightness of the mobile device. These factors are mitigated as much as we can by closing the services available in the device as well as decreasing the brightness of the device to be minimal. Furthermore, the loading time, fp and fcp measurements are collected through using an additional server which keeps listening for the measurements sent from the browser. This might cause a slight bias on the measurements collected since an additional server is executed throughout the duration of the experiment. Future replications of the experiment using different tool to collect the metrics will mitigate the potential threat to internal validity.

9.3 Construct Validity

The construct validity addresses the possibility of having an improper defined construct before translating them into measures. To mitigate this threat, we defined our constructs and measures using a GQM approach as seen in Chapter 4.

Mono-operation and Mono-method biases are two threats to construct validity. For the experiment, we have a single factor (JavaScript dead code) which is the only independent variable in our experiment design as provided in Chapter 5. We mitigate these biases by performing 20 trial runs for each web application of treatment.

9.4 Conclusion Validity

The conclusion validity addresses the statistical correctness and the relationship between treatment and outcome of the paper. The first threat for conclusion is having low statistical power. This threat is mitigated by performing 20 replication for each treatment subject. This will allow our statistical test to produce a more reliable result.

Furthermore, a threat for conclusion validity is the violation of statistical analysis assumptions. To mitigate this threat, we perform statistical test to address the the normality assumption. Additionally, the threat for error rate of the results needs to be addressed. To mitigate the error rate, we adjusted the significance level of the Dunn test by using the Bonferroni correction.

10

Related Work

At the time of writing this paper two papers were found addressing the impact of removing JavaScript dead code on certain metrics of mobile web applications.

Azim Afroozeh (10) conducted an experiment to determine the impact of dead code on Loading time and energy consumption of 16 real mobile web applications. The result of the impact of loading time indicated that there is no loading time difference in the original subjects with the presence of dead code compared to the adapted subjects with removed dead code. However, the result of the impact of energy consumption indicated that energy consumption is negatively impacted when using a lazy loading method compared to the original variant of the subject and to other dead code removal methods.

Shuxin He(13) conducted an experiment on 20 mobile web application to analyze the impact of loading time and energy consumption on dead JavaScript removal using level2 optimization level method of LacunaV2 tool compared to the original variant of the subjects. The result of the experiment indicated that energy consumption and loading time are not impacted by dead JavaScript removal using level2 optimization level approach.

11

Conclusion and Future Work

JavaScript dead code removal improves most of the metrics collected throughout the experiment. The experiment provides an understanding about the impact of JavaScript dead code on various metrics. The experiment is performed to compare the original code of each subject to the same code processed by the three different methods(optimization levels) of code removal.

The result of our experiment shows that the presence of JavaScript dead code negatively impact many metrics of the mobile web application including energy consumption, loading time, CPU usage, packets transferred and bytes transferred. However, render time such as FP and FCP are not impacted by the presence of JavaScript dead code. Additionally, the statistical tests for memory consumption show a significant difference between all treatment, however shows no significant difference when compared to the original treatment OL0. Therefore, we recommend developers to use the results of our experiment to gain insights on which run-time properties of the mobile web application are impacted by JavaScript dead code. Furthermore, the results help the developers understand that while developing a web application they should use and import only the wanted functions from a JavaScript framework/library. The presence of these unused functions will negatively impact various run-time properties of the web application.

The experiment does not provide a full verdict for all JavaScript dead code removal processes. The following are potential future work:

- Use LacunaV2 tool to remove JavaScript dead code using different analyzers combinations such as using nc+dyn+tajs which has the same fscore as the analyzer combination used in our experiment.

- Perform similar experiment with different and larger set of mobile web applications to verify the impact of the metrics. It will help provide a more clear representation of output for metrics such as memory consumption.

12

Appendix

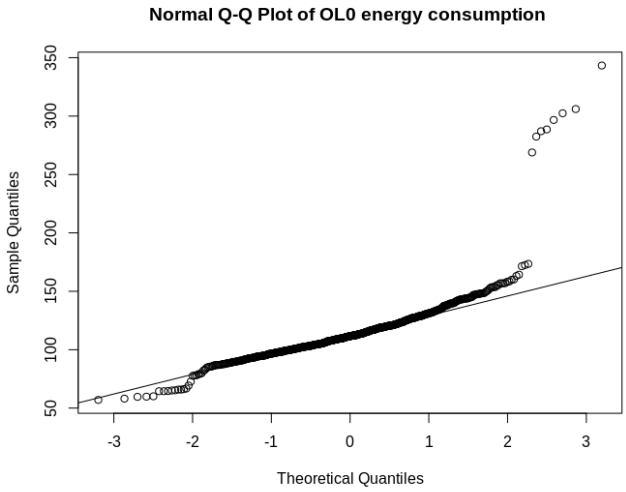


Figure 12.1: Normal QQ plot of OL0 Energy consumption

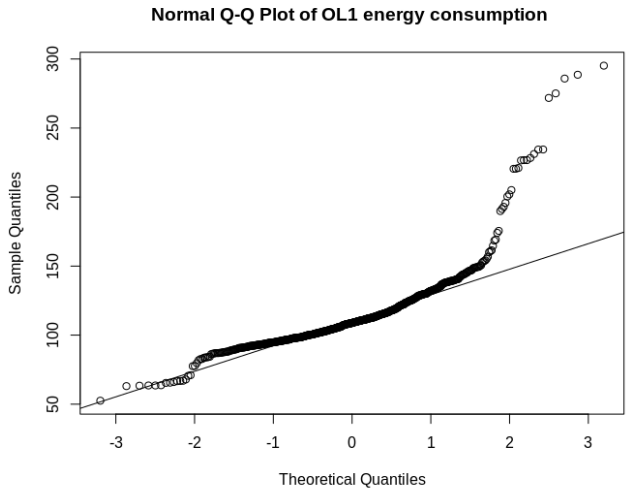


Figure 12.2: Normal QQ plot of OL1 Energy consumption

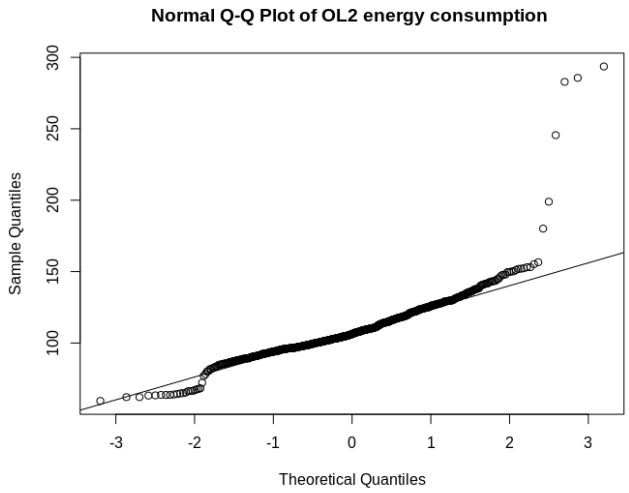


Figure 12.3: Normal QQ plot of OL2 Energy consumption

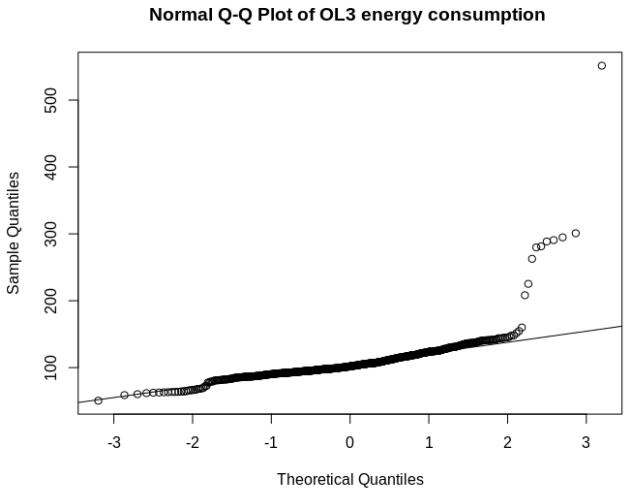


Figure 12.4: Normal QQ plot of OL3 Energy consumption

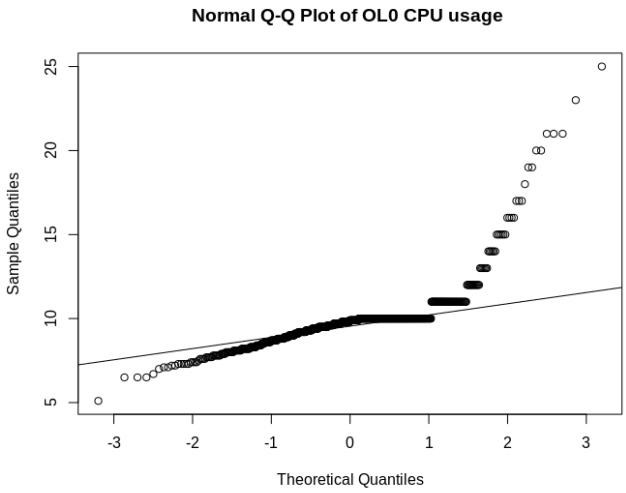


Figure 12.5: Normal QQ plot of OL0 CPU Usage

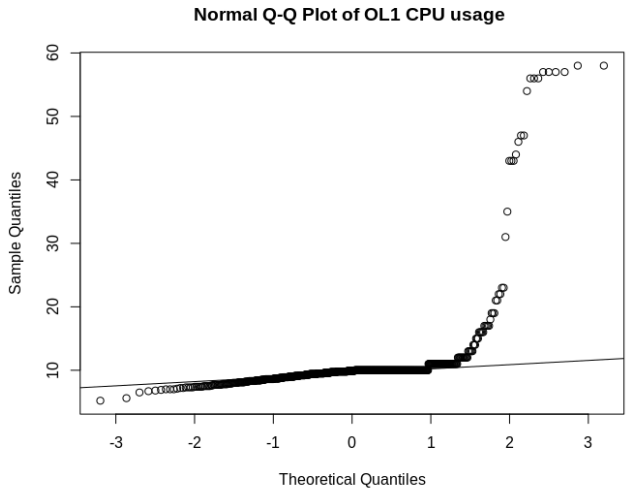


Figure 12.6: Normal QQ plot of OL1 CPU Usage

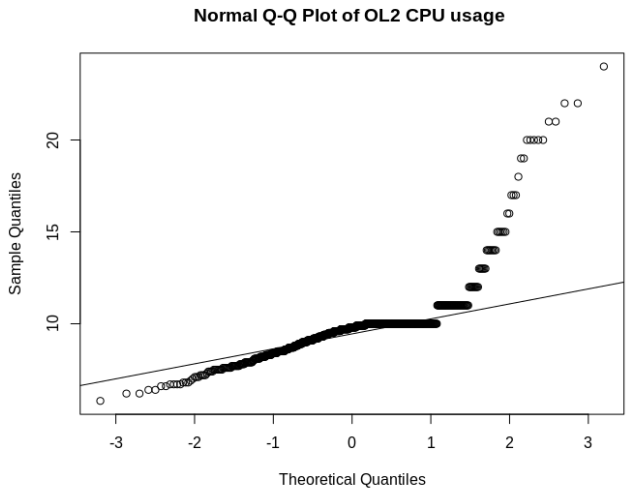


Figure 12.7: Normal QQ plot of OL2 CPU Usage

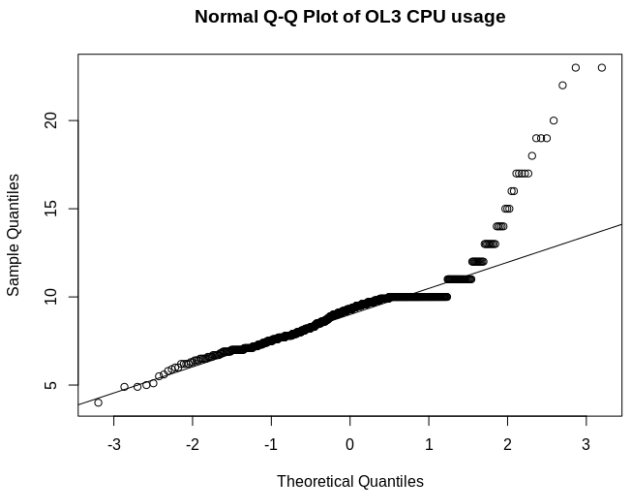


Figure 12.8: Normal QQ plot of OL3 CPU Usage

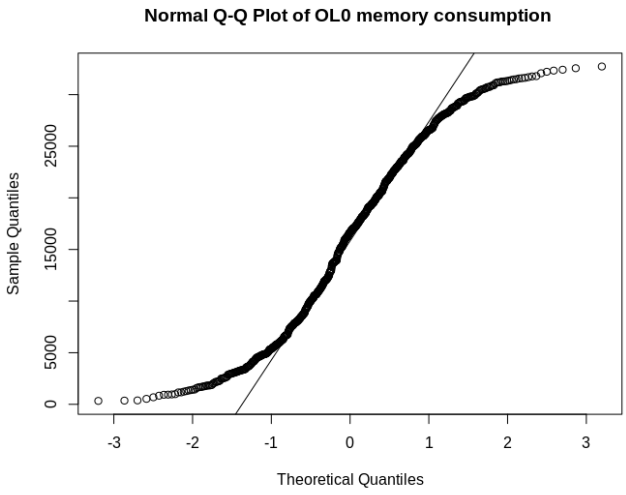


Figure 12.9: Normal QQ plot of OL0 Memory consumption

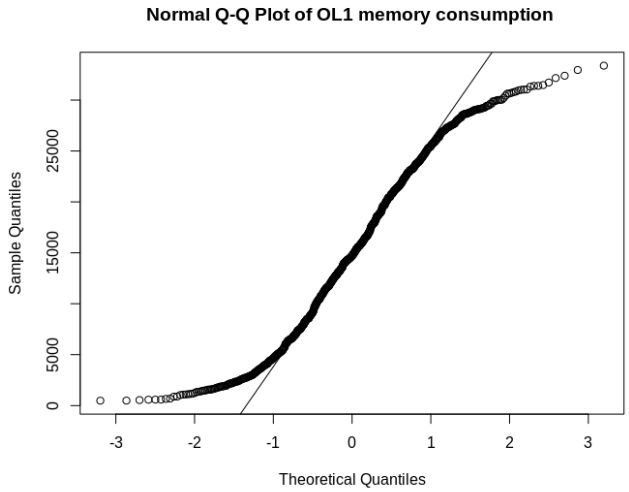


Figure 12.10: Normal QQ plot of OL1 Memory consumption

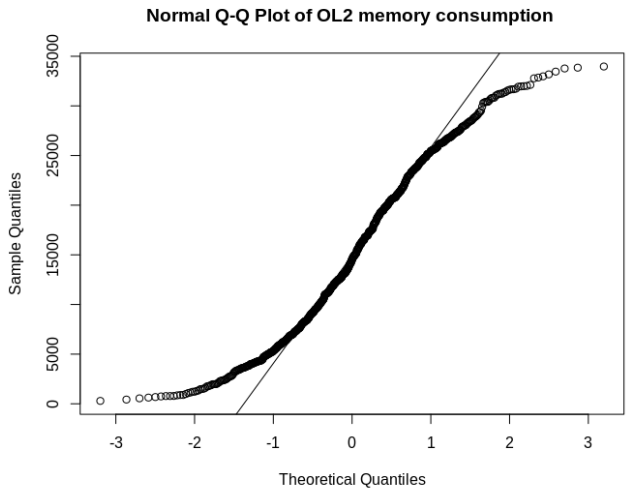


Figure 12.11: Normal QQ plot of OL2 Memory consumption

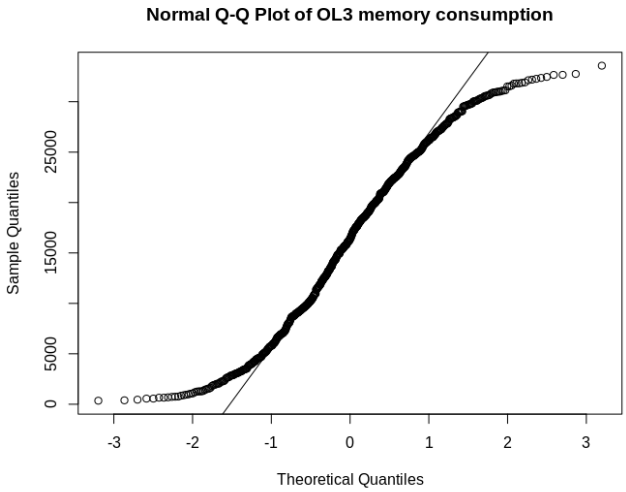


Figure 12.12: Normal QQ plot of OL3 Memory consumption

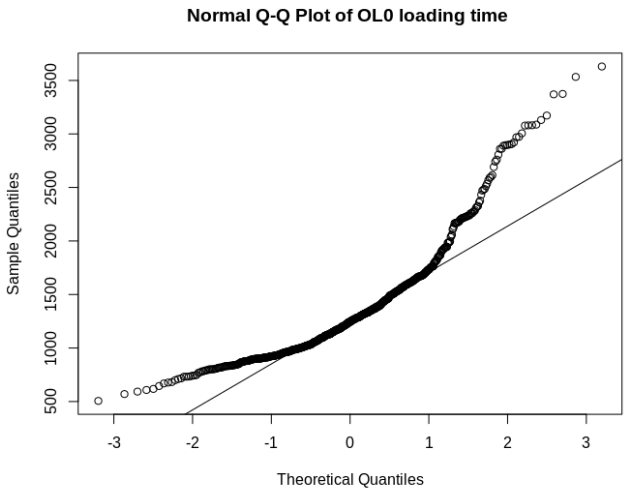


Figure 12.13: Normal QQ plot of OL0 Loading Time

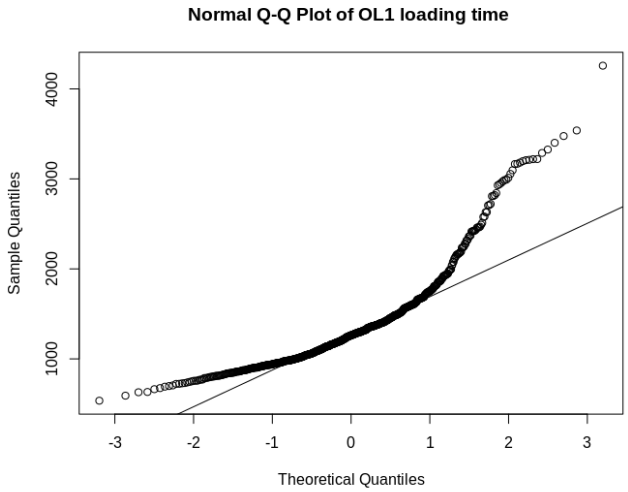


Figure 12.14: Normal QQ plot of OL1 Loading Time

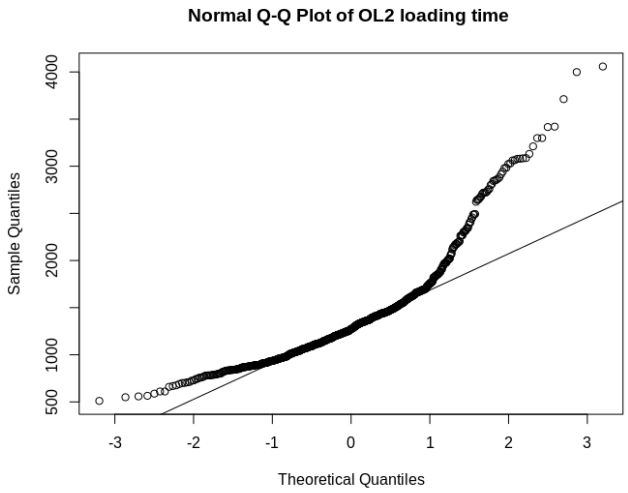


Figure 12.15: Normal QQ plot of OL2 Loading Time

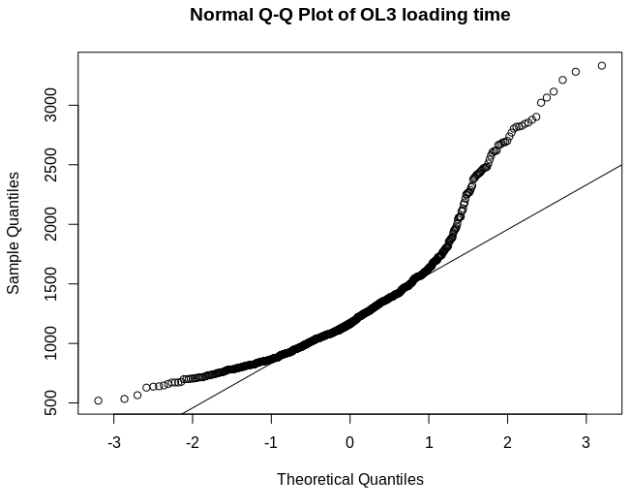


Figure 12.16: Normal QQ plot of OL3 Loading Time

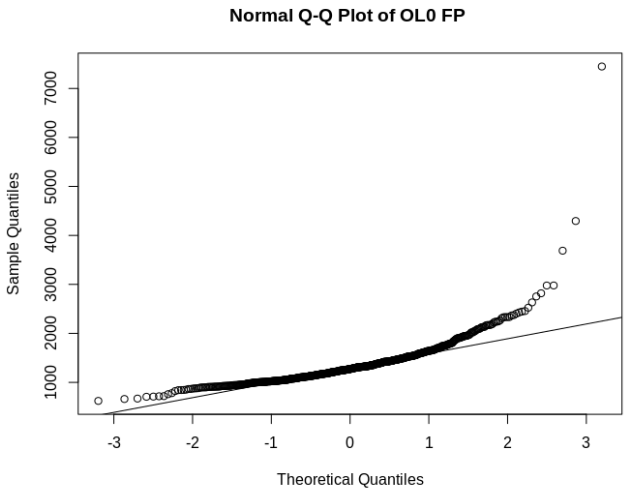


Figure 12.17: Normal QQ plot of OL0 FP

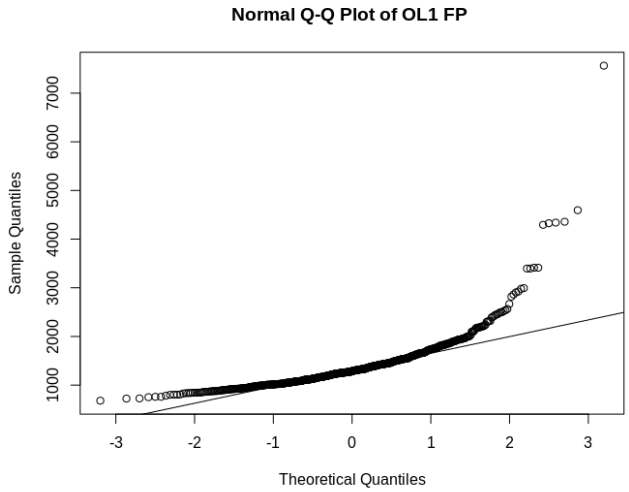


Figure 12.18: Normal QQ plot of OL1 FP

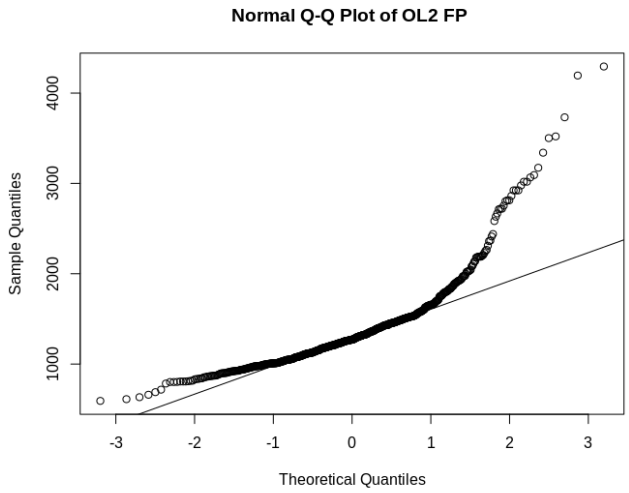


Figure 12.19: Normal QQ plot of OL2 FP

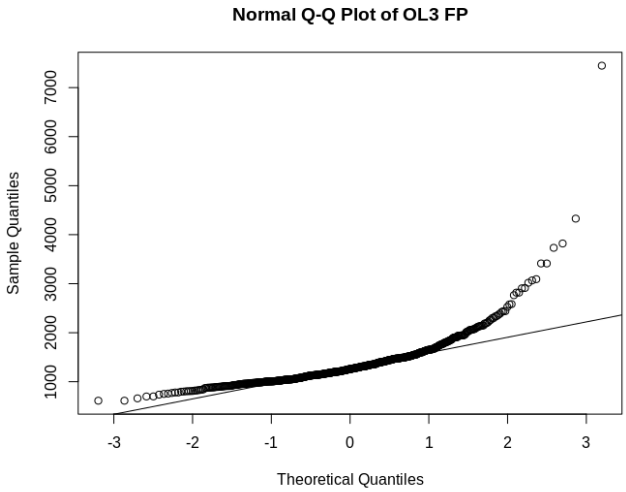


Figure 12.20: Normal QQ plot of OL3 FP

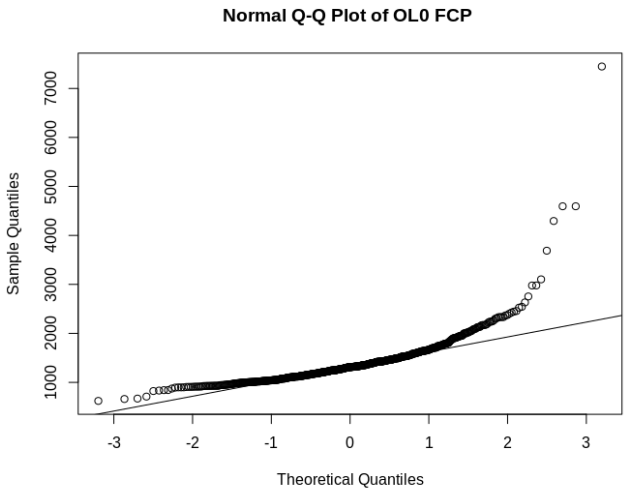


Figure 12.21: Normal QQ plot of OL0 FCP

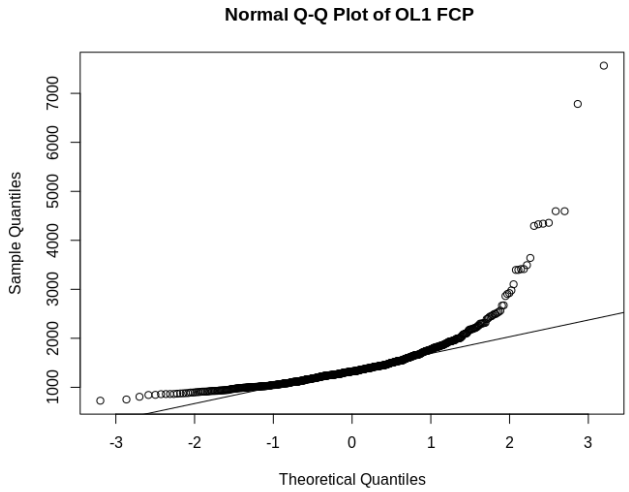


Figure 12.22: Normal QQ plot of OL1 FCP

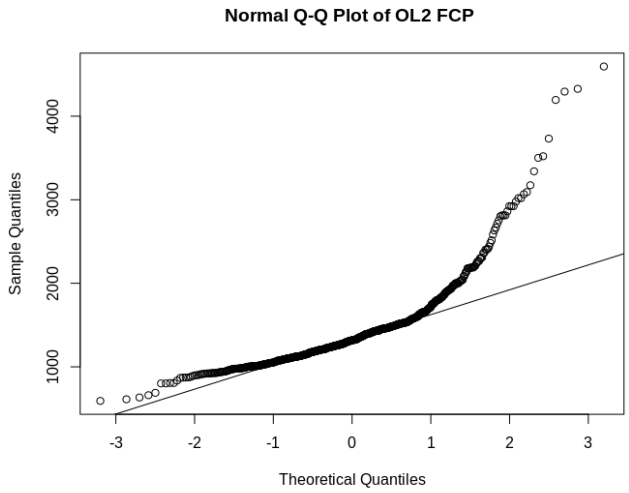


Figure 12.23: Normal QQ plot of OL2 FCP

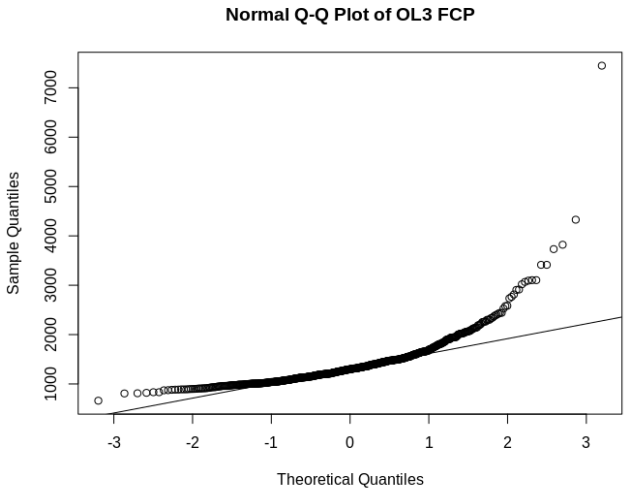


Figure 12.24: Normal QQ plot of OL3 FCP

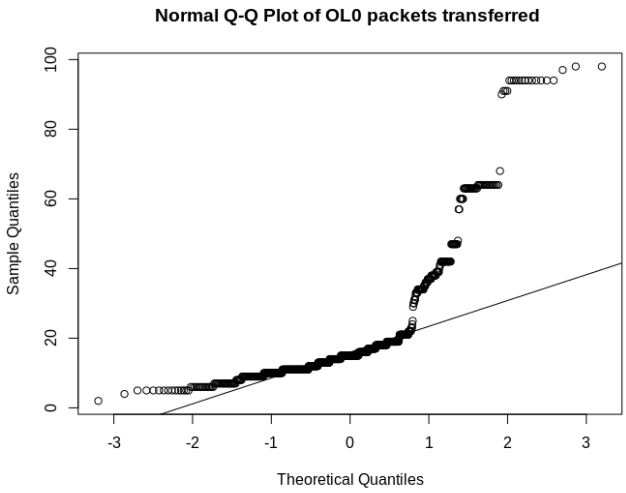


Figure 12.25: Normal QQ plot of OL0 Packets transferred

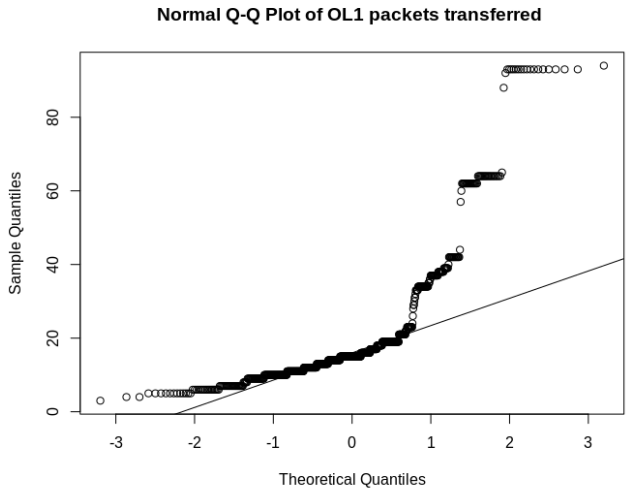


Figure 12.26: Normal QQ plot of OL1 Packets transferred

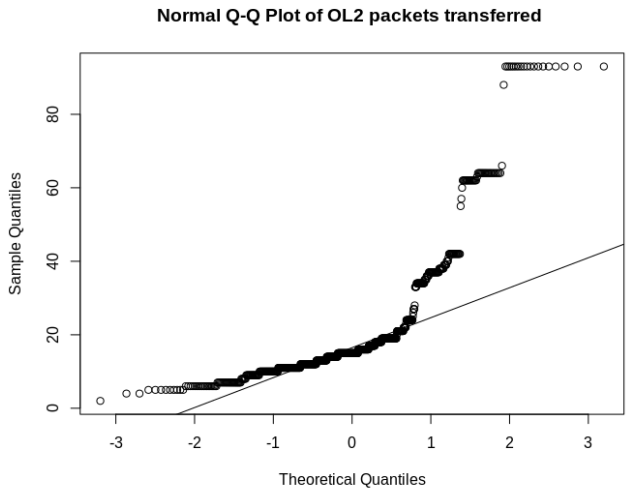


Figure 12.27: Normal QQ plot of OL2 Packets transferred

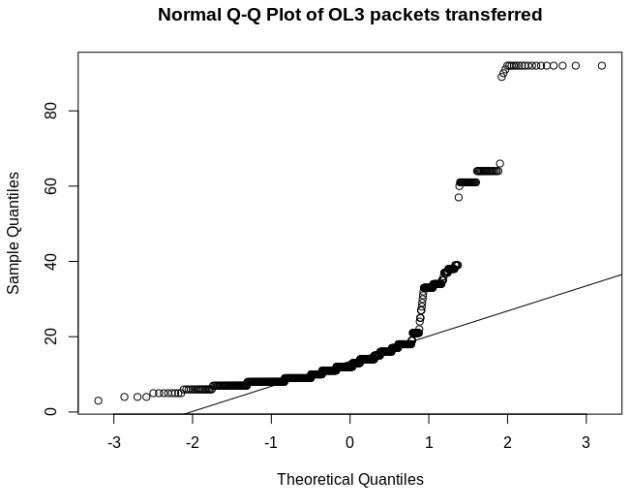


Figure 12.28: Normal QQ plot of OL3 Packets transferred

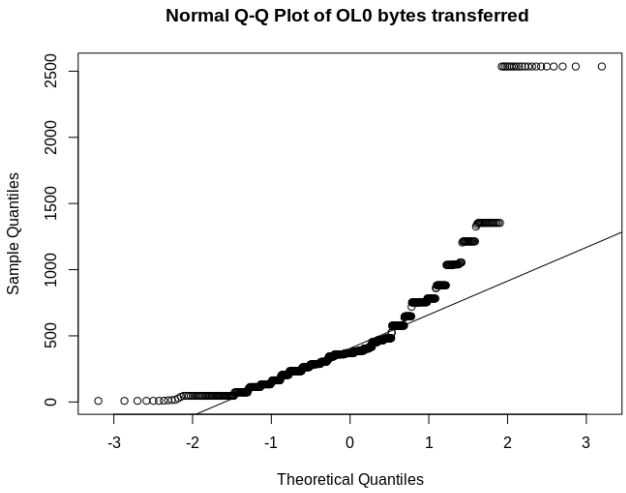


Figure 12.29: Normal QQ plot of OL0 Bytes transferred

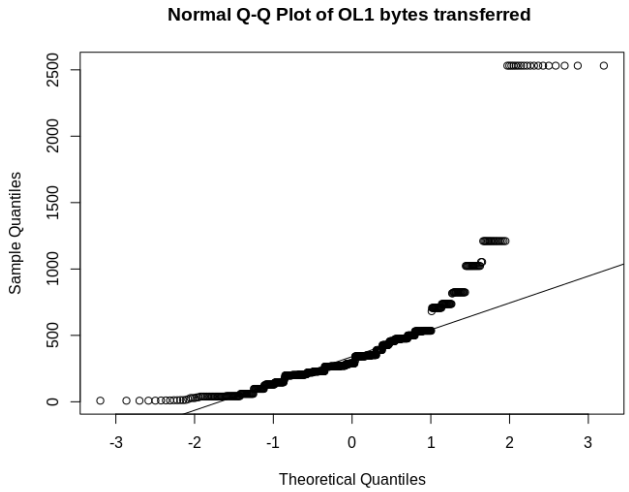


Figure 12.30: Normal QQ plot of OL1 Bytes transferred

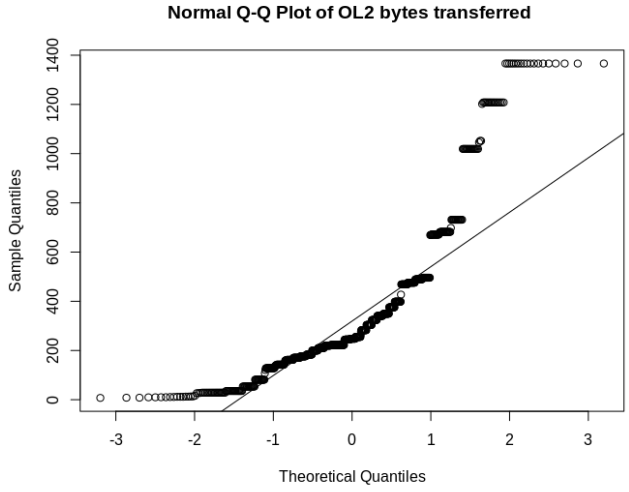


Figure 12.31: Normal QQ plot of OL2 Bytes transferred

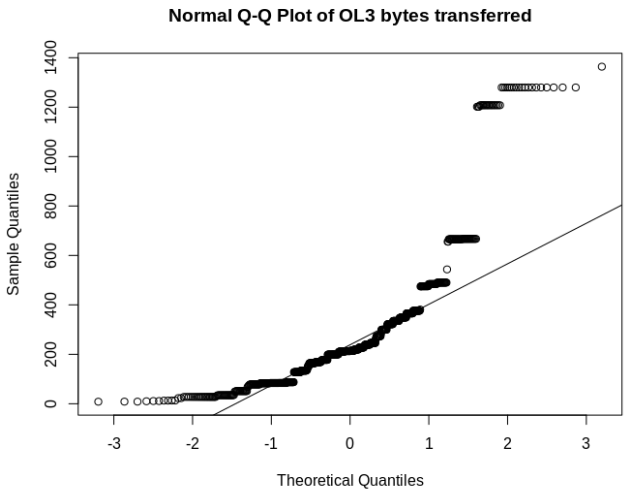


Figure 12.32: Normal QQ plot of OL3 Bytes transferred

References

[1] Niels Groot Obbink, Ivano Malavolta, Gian Luca Scoccia†, and Patricia Lago. **An Extensible Approach for Taming the Challenges of JavaScript Dead Code Elimination**. 2018. 1, 7

[2] Ivano Malavolta. **Beyond Native Apps: Web Technologies to the Rescue! (Keynote)**. In *Proceedings of the 1st International Workshop on Mobile Development*, New York, NY, USA, 2016. Association for Computing Machinery. 1

[3] H. Xi. **Dead code elimination through dependent types.** In International Symposium on Practical Aspects of Declarative Languages, 1999. 1

[4] J Adam Butts and Guri Sohi. **Dynamic dead-instruction detection and elimination**. *ACM SIGPLAN Notices*, **37**(10):199–210, 2002. 1

[5] Hyukwoo Park, Myungsu Cha, and Soo-Mook Moon. **Concurrent JavaScript Parsing for Faster Loading of Web Apps**. *ACM Trans. Archit. Code Optim.*, November 2016. 5

[6] A. M. Fard and A. Mesbah. **JSNOSE: Detecting JavaScript Code Smells**. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125, 2013. 5

[7] Kishan Nirghin. **Dead Code Identification, Elimination, and Assessment**. 2019. 6, 16

[8] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2012. 9

[9] Victor R Basili. **Software modeling and measurement: the Goal/Question/Metric paradigm**. Technical report, 1992. 9

[10] Azim Afroozeh, Saliha Tabbassum, Thijmen Kurk, and Wesley Geniz Shann. **Empirical Analysis of JavaScript Dead Code in the Wild**. 2019. 15, 55

[11] Hervé Abdi. **Coefficient of variation**. *Encyclopedia of research design*, **1**:169–171, 2010. 37

[12] Olive Jean Dunn. **Multiple comparisons among means**. *Journal of the American statistical association*, **56**(293):52–64, 1961. 43

[13] Shuxin He, Junfu Chen, Kiran Tjikhoeri, Shuqi Yan, and Omar de Munk. **Impact of dead JavaScript code in mobile Web apps**. 2019. 55