JSCleaner: De-Cluttering Mobile Webpages Through JavaScript Cleanup

Moumena Chaqfeh
NYU Abu Dhabi
Abu Dhabi, UAE
moumena@nyu.edu

Yasir Zaki
NYU Abu Dhabi
Abu Dhabi, UAE
yasir.zaki@nyu.edu

Jacinta Hu
NYU Abu Dhabi
Abu Dhabi, UAE
jh5372@nyu.edu

Lakshmi Subramanian
New York University
NY, USA
lakshmi@cs.nyu.edu

ABSTRACT

A significant fraction of the World Wide Web suffers from the excessive usage of JavaScript (JS). Based on an analysis of popular webpages, we observed that a considerable number of JS elements utilized by these pages are not essential for their visual and functional features. In this paper, we propose JSCleaner, a JavaScript de-cluttering engine that aims at simplifying webpages without compromising their content or functionality. JSCleaner relies on a rule-based classification algorithm that classifies JS into three main categories: non-critical, replaceable, and critical. JSCleaner removes non-critical JS from a webpage, translates replaceable JS elements with their HTML outcomes, and preserves critical JS. Our quantitative evaluation of 500 popular webpages shows that JSCleaner achieves around 30% reduction in page load times coupled with a 50% reduction in the number of requests and the page size. In addition, our qualitative user study of 103 evaluators shows that JSCleaner preserves 95% of the page content similarity, while maintaining nearly 88% of the page functionality (the remaining 12% did not have a major impact on the user browsing experience).

CCS CONCEPTS

• **Information systems → World Wide Web**; *Information systems applications*.

KEYWORDS

JavaScript, User Experience, Classification, Web Simplification

ACM Reference Format:

Moumena Chaqfeh, Yasir Zaki, Jacinta Hu, and Lakshmi Subramanian. 2020. JSCleaner: De-Cluttering Mobile Webpages Through JavaScript Cleanup. In *Proceedings of The Web Conference 2020 (WWW '20), April 20–24, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3366423.3380157

1 INTRODUCTION

According to a recent study [12], webpages that do not load in 3 seconds have the probability of losing their visitors increased to 32%. This probability is increased to 90% if a webpage does not load in 5 seconds. However, the current status of the web indicates that website owners are not yet carefully considering the Page Load Time (PLT), since the average PLT for an average mobile landing page is 22 seconds. In fact, the World Wide Web has witnessed a significant increase in webpages' size and complexity during the last decade. A modern web browser is required to typically process complex steps to load a single page including: (a) downloading 100+ objects [7]; (b) spawning 30+ network connections [10, 23]; (c) issuing 20+ DNS requests [23]; (d) processing JavaScript [20] along with several layers of recursive requests and HTTP redirections [7].

In trying to investigate the different factors that are affecting the overall PLT, two major factors can be distinguished: the networking cost of downloading the required resources, and the cost of processing these resources. From 2011 to 2019, the average total number of requests per page has increased by 23%, which results in a 300% increase in the total average requests' download size [3]. On the other hand, the dominant category in browser processing is proven to be JavaScript (JS) [20]. Taking the CNN home page as an example, and using the webpagetest [6], it is shown that JS requires 76.8% of the total Chrome browser processing time (assuming a medium-end phone such as Samsung Galaxy S7).

The cost of JS is currently handled by utilizing uglifiers or integrating browser extensions. With JS uglifiers, the size of JS files can be reduced to enhance the transmission efficiency, but the browser is forced to interpret the entire JS, thus sacrificing performance. On the other hand, the integration of a JS blocker as a browser extension can significantly reduce the cost of JS, either by blocking specific domain names or by disabling JS for a given host. The drawback in this case is that JS filename, domain name or host URL has to be explicitly specified by the user. There exist some extensions that aim to block a specific class of JS, such as Ad-Blockers. These blockers rely on a predefined list of domain names, which might be periodically updated. The limitation here is that there is no way of automatic classification for unknown JS.

In this paper, we micro-analyzed 500 popular webpages and quantified the types of JS they use. We identified that about 38% of the used JS are non-essential to the overall page aesthetics or functionality. We observed that another 26% of the used scripts can simply be translated directly to HTML, without having the end-user incurs the extra cost of downloading and processing these scripts. Inspired by these findings, we designed *JSCleaner*, a JS cleanup engine that aims at simplifying modern webpages by optimizing the JS usage. In contrast to other state-of-the-art approaches that eliminate the use of JS, block a list of known scripts, or redesign webpages from scratch, JSCleaner simplifies the use of JS in existing pages. It achieves this by selectively removing or replacing a portion of these scripts, rather than eliminating them. JSCleaner optimization process relies on classifying JS into three main categories:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WWW '20, April 20–24, 2020, Taipei, Taiwan
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7023-3/20/04.
https://doi.org/10.1145/3366423.3380157

non-critical, replaceable, and critical. Non-critical JS elements are those that do not directly enrich the end-user browsing experience. Typical examples are tracking elements, which are added to webpages mainly to help content providers generate revenue or create statistics. JSCleaner aims to reduce JS cost, by preserving only what is essential to the page content or functionality. The classification in JSCleaner is based upon analyzing the APIs utilized by JS to access the Document Object Model (DOM). We built an extensive and complete set of distinct API features and mapped each of them individually into a specific category. JSCleaner works for both: mobile and desktop pages. However, we choose to focus on mobile due to two main reasons. First, because mobile phones are now popular devices to access the web. Second, mobile phones can pose processing limitations that affect the browsing experience. In summary, this paper makes the following key contributions:

- It presents JSCleaner as a solution to reduce the utilization of client-side JS in modern webpages for an improved performance. It is powered by a novel rule-based classification engine that classifies JS based on 1262 distinct features.
- JSCleaner simplifies webpages and enhances the user experience without sacrificing the webpages' content and functionality. With the aid of a user study, we demonstrated that the simplified pages have about 95% content similarity to the original pages. In addition, about 88% of the page functional elements were retained. The remaining 12% did not have a major impact on the user experience since users reported a much higher similarity to the original page, and none of the core features were lost across any of the pages.
- Through the utilization of a diverse set of 500 popular webpages, we show that JSCleaner achieves a 30% reduction in PLT across a variety of network conditions and mobile phone types. Additionally, it reduces the number of network requests and page size by about 50%.

2 MOTIVATION

We motivate the need for JSCleaner, by highlighting the key issues of JS in today's web:

2.1 Javascript imposes a huge burden on PLT

Despite that modern web browsers have been improved for faster JS parsing and compilation, the cost of JS remains a key factor that imposes a huge burden on PLT. In [20], the cost of JS is quantified in example cases of popular webpages. Despite the cost of JS, an analysis of 2275 popular webpages shows that 29 JS elements are utilized at the 50th percentile, as shown in Table 1, whereas 72 lines of code per script are presented. Moreover, the 75% of the population shows around 80% increase in the number of scripts per webpage, compared to the median value. More surprisingly, the 75% shows around 1400 lines of code per script. These numbers demonstrate the massive utilization of JS in today's web.

2.2 JS worsens PLT in low bandwidth networks

To investigate the cost of JS in low bandwidth networks, we analyzed the performance of CNN.COM (which is heavily utilizing JS) in 2G nework settings with different phone devices using webpagetest [6]. Results showed that the high-end phone (iPhone 8 iOS 12) requires 7.6 seconds for processing JS, which represented 87.4% of the whole processing time. On the other hand, the emulated low-end phone (Motorola Moto G4) required more than 58 seconds for JS processing, which represented 97.4% of the browser processing time. For bandwidth-constrained users, the increasing cost of JS downloading combined with poor connectivity and low-end processing are resulting in a non-interactive web experience [27].

Table 1: JS Statistics of 2275 Poplular Webpages

Metric	25%	50%	75%	Maximum
Lines Per Script	12.0	72.0	1399.75	203532
Scripts Per Page	14.0	29.0	53.0	410

2.3 JS might not be essential

We micro-analyzed 500 popular webpages and quantified the types of JS elements they utilize. We identified that 38% of the used elements are non-essential to the page aesthetics or functionality. We observed that another 26% of them can simply be translated directly to HTML, without having the end-user incurs their downloading and processing cost. A tracking JS is an example of a script that is not essential to the page content or functionality. When these tracking snippets are injected in a webpage, they track the user activities such as the total spent time or the internal clicked links. Despite their benefit to content providers, injecting tracking scripts might worsen the PLT (especially for bandwidth-constrained users) without having an essential role from the user perspective. We expect a huge benefit from removing JS elements that are not essential to the page content or functionality, and translating the replaceable elements with their HTML counterparts.

2.4 Existing repetitions across JS

We utilized Stanford Moss -which is a system for detecting software Similarity- to check the similarity among different JS files embedded on the same page. We randomly picked few cases from the set of popular webpages, and surprisingly, we have found websites where different independent JS files are almost identical. For example, americanbar.org has four different independent JS files with 99% similarity. Another example case is bhg.com, where two independent JS files were found to have 92% similarity.

3 RELATED WORK

Recently, there has been an increasing interest in the research community as well as the industry to tackle complexity issues at the page level in today's web. For example, SpeedReader [11] aims at enhancing content-rich webpages that are suitable for the reader mode of web browsers. Unfortunately, SpeedReader is not capable of enhancing the performance of non-readable pages that utilize JS for content generation. Wprof [24] is another in-browser tool that acts as a profiler to provide an understanding of the key hindering effects behind PLT. It builds a dependency graph between the different browser components, and shows that JS has a considerable impact on the PLT due to the role it plays in blocking HTML parsing. To address the inefficiencies in the page loading process,

Shandian [25] was proposed. It restructures the page loading process to speed up the PLT, by preloading the page and sending an initial DOM to the user. Although it enables faster PLTs, Shandian keeps the entire page elements without attempting simplification. Polaris [18] is another tool that tracks data flows during the page load, which in turn detects additional edges compared to existing dependency trackers. Since these edges allow for more accurate fetch schedules, they can contribute in reducing PLT. Our work is inspired by Polaris, where the overall PLT can be enhanced by reducing JS complexity in modern webpages. In addition to PLT improvement, different solutions have been proposed to improve the user browsing experience [19]. From an industry standpoint, both Google and Facebook have attempted to tackle today's web complexity through Google AMP [13] and Facebook Lite [2]. AMP redefines how pages should be written, by providing web developers with a framework to create their webpages. A major difference between our approach and AMP, is that we aim at simplifying what already exists in today's web rather than creating new webpages. On the other hand, Facebook Lite is an application designed for Android and iOS mobile phones, which can perform effectively on all networks using low-end phones. In contrast to JSCleaner that provides a generalized framework for web content simplification, Facebook Lite is designed for Facebook applications.

Despite that JS plays a significant role in webpages performance, and although it became one of the most popular programming languages, its practical performance was rarely examined. A recent study [22] shows that inefficient APIs usage is the most common cause of JS performance issues. In [8], the authors present JSExplian, which is a JS reference interpreter that produces execution traces, in order to interactively investigate these traces in a web browser with conditional breakpoints. However, JSExplain does not handle unspecified JS browser behaviors. In addition, its formalization includes only the syntactic rules, but not the parsing rules of JS. A formal semantic parser can be found in [21], whereas an operational JS semantic parsing is proposed in [15]. In a recent work [26], the authors performed an empirical study for the non-deterministic behavior of JS under certain conditions. In contrast with [22], [8] and [26], and instead of examining the exact behavior of JS, we propose to reduce the number of JS elements utilized in modern webpages without sacrificing the pages' contents and functionalities. The desired reduction is achieved by classifying JS elements to make a decision on preserving, removing or replacing them with HTML. Our classification is based on detecting the DOM APIs and the HTML DOM APIs in JS. In that sense, we provide a web-oriented approach to understand JS functionality, with the objective of performance and user experience improvement.

4 JSCLEANER

The main goal of JSCleaner is to enhance the web browsing experience, by transforming webpages to simpler or "lighter" versions. The rationale behind the design of JSCleaner is to optimize JS usage in today's web by classifying different JS elements in webpages, and then make a decision on preserving, translating, or eliminating them. The optimization is achieved by utilizing a smaller portion of JS resources that were embedded in the original pages for performance improvement, while maintaining a high similarity to the original pages in terms of content and functionality. For a webpage to be simplified by JSCleaner, both inline and external JS elements are required to be extracted. Inline scripts are found within <*script*> tags in the page HTML source, whereas external scripts are fetched from external resources identified by the "src" attribute of the corresponding <*script*> tag. JS extraction refers to the process of fetching JS code. Figure 1 shows the generalized data flow diagram of JSCleaner, where the main processes are shown with their input/output. As the Figure shows, a webpage of a given URL is inputted to the script extraction process, which outputs a set of JS elements that were embedded in the webpage. These elements are inputted to the feature extraction process. JS code is parsed to extract a list of features and feed a rule-based classification algorithm (4.2), which attaches a class to each JS element based on its features. Feature extraction and rule-based classification are aided by a feature store (4.1). Finally, the page is tackled by a simplifier (4.3), that decides on each JS based on its class before outputting the simplified webpage.

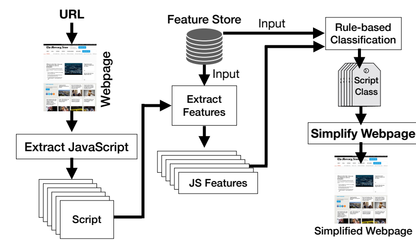


Figure 1: JSCleaner data flow diagram

Analyzing JS with the aim of understanding their exact behavior is a challenging and time-consuming process. The complexity of JS might often lead experts to get puzzled by the output of a JS code segment [8]. Thus, there is a need for a different approach that is agnostic to the exact behavior of JS elements injected in webpages, but can still provide a high-level insight into them. Since webpages are typically HTML documents, the design of such an approach requires understanding how JS interacts with the Document Object Model (DOM), which is the main programming interface for HTML documents that defines their logical structure and the way in which they can be accessed and manipulated.

To interact with the DOM, JS utilizes a set of exposed Application Programming Interfaces (APIs). Therefore, and instead of trying to understand the exact behavior of a JS code segment injected in a webpage, it is meaningful to analyze the utilization of APIs in that segment to interact with the DOM. From a high-level point of view, JS interaction with the DOM can take one of the following forms: reading, writing, and/or event handling. Hence, we create a set of features based on the specifications of DOM APIs [16] and HTML DOM APIs [17] and map each feature to one of these interactions. The mapping process results in a complete feature store that consists of a full set of labeled features.

Our feature store aids a classification algorithm, which classifies each JS element found in a webpage to make simplification decisions on the page based on these classes. The classification algorithm prioritizes event handling as a feature that is critical to the browsing experience. If a script is found to utilize features that were labeled as event handlers, then it is classified as *critical*. Whenever a script

is found not to use event handling features, it is checked against the features labeled as writing features. Many modern webpages utilize these features to produce a set of HTML tags and attach them to the DOM. We refer to these scripts as *replaceable* scripts, in the sense that their content can be translated into pure HTML. Besides, any script that is found not to be critical or replaceable is considered as *non-critical*. To produce a lighter version of a webpage, JSCleaner employs two major processes:

(1) JavaScript Classification: is the process of categorizing each JS element in a webpage based on a set of features, into one of the following categories: critical, non-critical or replaceable.
(2) Webpage Simplification: is the process of making a decision on each of the classified JS elements to create the simplified page. The decision can be: preserve, translate, or remove.

4.1 Feature Engineering

The feature engineering process aims to create a set of features to aid JSCleaner in feature extraction, JS classification and webpage simplification. We define a feature as a JavaScript-Web (JSW) feature, which can be a property, a method or an event handler that can be accessed by JS via a Document Object Model (DOM) [16] API, or a DOM HTML [17] API. To design a complete set of features, we consider the full list of DOM APIs and DOM HTML APIs that are identified in [16] and [17], along with their properties, methods, and events. These APIs represent all possible ways to access an arbitrary webpage according to the latest DOM standards. These APIs are not part of the core JS language specification, but they are modeled as JS objects to provide access to webpages.

To ensure the completeness of the considered set of APIs, we selected a dataset of 2275 popular webpages, and extracted the API calls from the JS elements they utilize. The total number of JS elements found in these webpages is 89,998. Results show that there is no missing API in our list of DOM and DOM HTML APIs, since no additional APIs could be found in the extracted list of APIs. Therefore, our list of APIs can be considered as a complete set of interfaces to connect webpages to JS or programming languages.

We referred to the specification of each of the DOM and the DOM HTML APIs to manually label each of the JSW features based on the description that is provided by [16] or [17]. The full feature set includes all the properties, methods and events associated with the APIs. The manual feature labeling process aims at correctly categorizing each of the features in order to aid the JS classification and the webpage simplification processes. Based on the full set of JS APIs, we could identify four possible labels:

(1) A Reading Feature: where the only possible utilization of a feature is to read or extract information via an interface. For instance, the only way to utilize the method getElementsByTagName() of the document interface is to pass a tag name as a parameter (such as 'script' for instance), so that the required set of DOM elements is extracted and returned.
(2) A Read/Write Feature: where the same feature can be used to get (read) or set (write) values via interfaces. For example, the element.id property is used not only to return (read) the element's identifier (which reflects the id global attribute), but also to set the id value, which must be unique in a document.
(3) A Writing Feature: where a feature can only be used to alter HTML, by creating/adding new features, or changing/removing existing features. These include both the writing and the pre-writing properties and methods. The appendChild() method of the Node interface presents a clear example of a writing feature that writes to HTML by adding an element. It specifically adds a child node to the end of the list of children of a certain parent node. Since the only possible utilization of the appendChild() method is to write to HTML, it is labeled as a writing feature. An example of pre-writing features can be represented by the method document.createElement(), which can create an HTML element specified by a tag name, but do not actually attach the created element to the DOM.
(4) An Event feature: where a feature is correlated with a webpage event by any mean (creation, setting, handling, or removing). Example properties include onclick (which is used for processing click events on elements), and onmouseover (which is fired when the user moves the mouse over a particular element). On the other hand, a clear representative example method for an event feature is document.createEvent().

Prior to deciding the final feature set, we reviewed the full set to identify duplicate features, where a property or a method is found in the specification of more than one interface. These features include common properties and methods such as id, name, type, width, length, and height. From the design point of view, these features can be neglected due to their common utilization, even by user-defined functions and the JS language core [14]. After discarding duplicates, we end up with a feature store of 1262 labeled features. These features can be reproduced by extracting all the APIs specified in [16] and [17], along with their properties, methods, and events.

4.2 JavaScript Classification

JS classification aims to attach a particular class to each JS element found on a webpage. To do so, JSCleaner employs a feature extraction process to generate a list of features for each script injected in a webpage. Whenever a DOM or HTML DOM access is captured in a script, its features list is updated to reflect that access. Feature extraction is based on the labeled JSW features located in the feature store. Each JS element injected in a webpage is parsed by JSCleaner to extract all the features that are found to be identified in the JSW feature store. As we show in Figure 1, the output of the feature extraction process is a set of records, each record represents the features extracted from a certain JS. For a given webpage, the total number of records generated by the feature extraction process equals the number of JS elements found in the given webpage. These records of extracted features are inputted to the JSCleaner classification algorithm for one of the following class assignments:

(1) Critical: where a set of features labeled as events are found.
(2) Replaceable: where a set of features labeled as writing features are found.
(3) Non-critical: where neither event features nor writing features are found.

Specifically, we classify JS elements that utilize event features as critical due to the fact that the main objective of linking webpages to JS is to provide user interactivity features, such as displaying

Table 2: Rule-based classification symbols

Symbol	Interpretation
$F(x,d)$	x is a feature of document d
$l(x)$	the label given to feature x
$f(d)$	the class assigned to document d
$u(x,d)$	the utilization type of feature x in document d
e	an event label
w	a writing label
rw	a read/write label
$critical$	Critical JS class
rep	Replaceable JS class
$noncritical$	Non-critical JS class

a list of options by clicking on a drop-down menu item. Event handling is the only known methodology to provide such a user interactivity feature in webpages. Besides event handling, JS can be utilized to dynamically produce and attach HTML to webpages via writing features. We refer to these scripts as replaceable scripts, in the sense that their content can be translated into pure HTML. Any script that is not critical or replaceable is considered as non-critical from the user perspective. The rationale behind the classification approach of JSCleaner is to reduce JS usage in webpages, by making simplification decisions for each JS element based on its class.

Formally, we define a domain of JS documents $D = \{d_1, d_2, ..., d_n\}$. The classification function $f : D \rightarrow C$ assigns a class c from the set of predefined classes $C = \{critical, replaceable, noncritical\}$ to each document $d_i \in D$. The class assignment is based on a set of predefined logical rules. The classification function considers a list of extracted script features to classify each script according to these rules. Each rule can be expressed as a predicate logic sentence. Table 2 provides the symbols interpretation of the rules. The basic logical rules considered in our classifier is represented as following:

$$\exists x : F(x,d) \land l(x) = e \rightarrow f(d) = critical \tag{1}$$

$$\begin{gathered}((\exists x : F(x,d) \land l(x) = w)\lor \\ (\exists x : F(x,d) \land l(x) = rw \land u(x,d) = write))\land \\ (\nexists x : F(x,d) \land l(x) = e) \rightarrow f(d) = rep\end{gathered} \tag{2}$$

$$\begin{gathered}(\nexists x : F(x,d) \land l(x) = w) \land (\nexists x : F(x,d) \land l(x) = e)\land \\ (\nexists x : F(x,d) \land l(x) = rw \land u(x,d) = write) \rightarrow f(d) = noncritical\end{gathered} \tag{3}$$

Our rule-based classifier starts by assigning a default class to the script being processed. Then, it evaluates the script extracted features to set the final class. The algorithm defines a list of feature labels $L = \{e, w, rw, r\}$, ordered according to predefined priority values. The algorithm has access to the JSW feature store to map each extracted feature to its corresponding stored label. According to the predefined priorities, the classifier looks for event features first. If a JS element is found with a feature labeled as e, it will be classified as $critical$ as stated in eq. 1. Otherwise, if no event features are found, the algorithm examines the next prioritized features that are labeled as w. If any of these features is fetched in a script, a $replaceable$ class is returned. Another case is identified where a JS element is classified as $replaceable$, which is found in the second line of eq. 2. When there exists a feature that is labeled as read/write feature (rw), the classifier checks the utilization function $u(x,d)$, which defines the utilization of feature x in a document d. Possible utilization of an rw feature is either to $write$ or $read$. According to eq. 2, if an rw feature is found with utilization $u(x,d) = write$, a $replaceable$ class might be returned (if the third clause of the equation applies). In cases where neither event features nor writing features are found in a script, it is classified as non-critical (eq. 3).

The classification algorithm considers JS dependency to identify and re-classify dependent scripts. These are defined as JS elements that depend on other JS to function properly. Hence, we propose dependency tracking to ensure that JS dependency does not result in broken page content or functionality. That is, grouping dependent elements, and re-classifying them when necessary. For a group of dependent JS elements, the re-classification checks the highest priority class among the group, and assigns the same class to all the elements in the group. Possible classes are ordered according to their priority, from the highest to the lowest as: critical, replaceable, and non-critical. One approach to implement such a JS dependency tracking is to utilize the reference errors reported in the browser console. This can be achieved by enabling one script at a time, and checking for reference errors. If an error exists, then we start enabling other scripts one at a time until that error disappears. Since this is an extremely costly process in terms of time complexity, we currently consider only a two-level dependency.

4.3 Webpage Simplification

Once every JS element injected in a webpage is assigned to the appropriate class, a web page is tackled by the simplification process, which makes a decision on each script based on its class. Every JS element that is marked as non-critical is completely removed from the new simplified page, whereas critical elements are preserved. For scripts that are classified as replaceable, we aim to translate them into HTML for the purpose of a complete replacement. We utilized the HTML output generated by those scripts using Selenium WebDriver [5], along with Firefox 67.0 (64-bit) browser.

For each webpage, JSCleaner prepares an intermediate version which only contains replaceable scripts, and opens it using Selenium to save its source. This way, JSCleaner can utilize Selenium for translating replaceable JS into pure HTML, and then remove them from the translated page. In handling replaceable JS, JSCleaner assumes the following versions of a webpage:

(1) *The original page*: indicates the original HTML of the page.
(2) *The page to translate*: is the HTML of *the original page* with critical/non-critical JS removed. Keeping only replaceable JS.
(3) *The translated page*: is the HTML source extracted after opening *the page to translate* using Selenium.
(4) *JSCleaner Page*: is the final version of the JSCleaner webpage, after removing replaceable JS from *the translated page*, and inserting back the critical JS found in *the original page*

In Figure 2, we show the steps involved in the simplification process. To generate the *JSCleaner Page* of a certain webpage, *the original page* is used to generate the corresponding *page to translate*, which is then opened using Selenium to translate replaceable

JS into HTML. The resultant *translated page* is then cleaned by removing replaceable scripts (since their HTML has already been translated), and to eliminate the potential redundancy that may occur due to the utilization of Selenium source. Finally, critical JS elements found in *the original page* are inserted back before outputting *JSCleanerPage*.

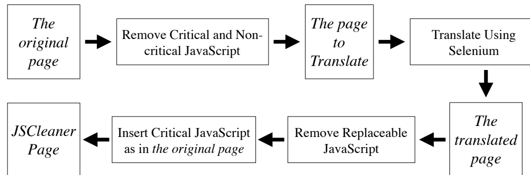


Figure 2: JSCleaner simplification process

5 IMPLEMENTATION AND DEPLOYMENT

Our prototype implementation consists of the JSCleaner engine and a proxy server both written in python, as well as a MySQL database. We selected a set of 500 popular webpages from [4], which includes diverse pages such as: news sites, education, sports, entertainment, travel, and government pages. The proxy server is used to clone and replay webpages for two main purposes: first, to extract the embedded JS, and second, to enable comparative performance evaluation and analysis. i.e., being able to freeze a particular version of a webpage to have reproducible results. We extended MITM proxy [9] using its flexible scripting approach. We created our own addon to intercept all HTTP requests, and create our own HTTP replies. By linking this addon to the database, it is possible to clone and replay HTTP replies with their full content. When an HTTP request comes in, the database is checked for a possible match to serve the cached HTTP reply immediately. If the request is not found, the script allows for the normal proxy procedure to fetch the request from the Internet. Once the reply comes in, and before serving it back to the client, a copy is saved locally and a new entry is stored in the database. JSCleaner follows the steps discussed in section 4 to produce the simplified JSCleaner version for each webpage, and then inject it into the proxy and the database, using the same original URL with the addition of "JSCleaner.html" concatenated to its end. To deploy JSCleaner, we envision three different practical scenarios:

- As a Proxy server: Similar to our current implementation scenario, JSCleaner can be deployed as a separate web proxy that is capable of delivering simplified versions of webpages either on the fly, or as a cached version. This could be in the form of a business enterprise that serves customers with simplified versions. The drawback is that it is not transparent to the clients, because they need to configure the proxy.
- As a Browser Plugin: JSCleaner can be integrated into a browser plugin to analyze and classify JS elements embedded in webpages, and then block certain elements based on their classes. To gain the full potential of this approach, the plugin must analyze each page at least once beforehand, to aid blocking certain non-critical JS as well as translate replaceable ones. This approach can benefit from a centralized server that the plugin can share its own classification outcome with regularly, as well as receive regular updates on webpages analyzed by others. This would help in decentralizing the classification process across multiple devices. Of course, certain considerations need to be addressed, such as versioning control and classification expiries.
- As an Analysis Engine for Content Providers: This scenario suggests to use JSCleaner by content providers as an analysis and a simplification engine. By exposing the different groups of replaceable and non-critical JS elements, content providers can decide to alter their existing webpages or offer lighter versions for performance improvement.

Each of the three deployment scenarios could work on its own. We envision that the most promising scenario is the browser plugin, since it does not require many changes to the network or content servers. In addition, it is relatively easy for web users to utilize browser plugin. Finally, JSCleaner can have a caching mechanism with a certain timeout strategy. If a page is still within that time, then the cached version can be served, otherwise, the new page would be re-created. In this paper, we considered running JSCleaner when the webpages are requested for the first time.

6 EVALUATIONS

The evaluation methodology focuses on JSCleaner performance from several aspects. We split our evaluations into two separate main categories: quantitative and qualitative evaluations. The aim of the quantitative evaluation is to highlight JSCleaner performance gains, whereas the qualitative evaluation aims at assessing JSCleaner accuracy in terms of the attained webpage similarity compared to the original, including both visual and functional similarities. We cloned a set of 500 popular webpages using the MITM proxy, which was also utilized to serve these pages for the purpose of evaluation. We compared the performance of two versions for each webpage:

- The original page.
- JSCleaner page, which is a simplified version of the original page, generated by JSCleaner.

6.1 Evaluation Setup

The evaluation setup consists of two desktop computers and a smart phone, as shown in Figure 3. We used two different phones for the evaluations: Xiaomi 6A, and Samsung Galaxy S8+. The Xiaomi phone is considered a low-end smart phone that costs about 80$, and comes with a 2GB RAM. The Samsung Galaxy S8+ is considered a high-end device, that costs about 500$ and comes with 6GB RAM. The first desktop machine is used as a web server that serves the 500 webpages, using the modified MITM proxy. We pre-cloned all the pages beforehand. The MITM proxy has been extended to replay contents by examining the request headers. The mapping metadata is saved in a database, where the MITM proxy first checks if the request URL exists, and then gets the saved object content from the hard disk and serves it back to the client. This MITM proxy was configured to operate on port 8080 to serve the original webpages. We configured a second MITM proxy to operate on the same machine (on port 9999) to serve the JSCleaner pages. The reason behind using a second proxy is that JSCleaner eliminates a number of non-critical scripts by removing them from the rendered

page. An eliminated script might not be part of the index (the page before rendering), where the script can simply be requested recursively by another resource from within the index page. If such a script request is detected by the MITM proxy, the proxy will simply reply with an empty object with status code *"200 OK"*.

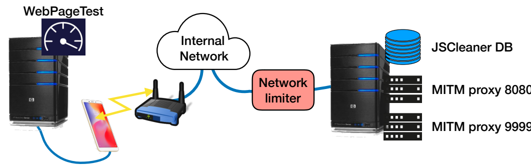


Figure 3: Evaluation setup

The second desktop machine is used for running the Webpagetest [6]. The Webpagetest tool is used to automate webpage requests on a mobile phone by firing one of the phone browsers and requesting a certain page. The tool hooks into the browser dev-tools and records various metrics such as the different page load timings ranging from the first paint to the full load time of the page. Webpagetest also records the full waterfall HTTP Archive (HAR) representation of the session as well as a screen-shot of the rendered page. We configured Webpagetest to request the 500 webpages using the Chrome web browser. The phone is connected to the 2nd Desktop machine using a USB cable so that the Webpagetest can control and record the experiments. On the other hand, the phone is connected to the first desktop machine using Wi-Fi. We alternate between the two proxies depending on the experiment, i.e., original vs. JSCleaner. In order to study the effect of JSCleaner, we evaluated 4 different network types between the phone and the proxy:

- Emulated 2G: 280 kbps bandwidth and 800 ms Round Trip Time (RTT).
- Emulated 3G: 1.6 Mbps bandwidth and 300 ms RTT.
- Emulated 4G: 9 Mbps bandwidth and 170 ms RTT.
- No network restrictions: using the phone Wi-Fi with no bandwidth restriction and RTT lower than 1 ms.

In terms of the evaluation metrics, we mainly focused on:

- First Meaningful Paint: is the time when the main element of the page is shown (known as the hero of the page).
- Dom Interactive: is the time when the DOM has finished analyzing all HTML and blocking JS objects.
- Dom Complete: is the time when all of the processing is complete and all of the resources finished downloading.
- Full Load: is the overall page load time when everything is fully loaded.
- Page Size: is the overall downloaded page size.
- Number of requests: is the total number of objects requested by the page over the network.

6.2 JSCleaner Quantitative Evaluation

To evaluate the effectiveness of JSCleaner in terms of speeding up PLTs, we compared the 500 webpages for both the original page and JSCleaner page. Since the idea behind JSCleaner is to remove a number of non-critical scripts and translate replaceable scripts with their HTML counterpart, we hypothesized that there should be a considerable improvement in PLTs, so that the utilized low-end mobile phone is not required to evaluate and process the same number of JS elements embedded in the original webpages. In our quantitative evaluation, We first quantified the gain of only cutting down JS processing without the effect of any other factor. This was achieved by simply not putting any restriction on the bandwidth or the network delay, hence the PLT will mainly be influenced by the JS processing cost. Second, we aimed to investigate if the network has an impact on JSCleaner evaluation due to the dependencies between JS browser processing and the network requests that can block each other [24]. Thus, we evaluated the performance over multiple cellular channels, mainly: 2G, 3G and 4G network. Since the first and the second quantitative evaluations were carried out using a low-end smart phone, the third evaluation revolved around testing JSCleaner gains when using a high-end smart phone.

6.2.1 Effect of JavaScript Processing. In this evaluation, we quantified the impact of JSCleaner by focusing only on the reduction of JS processing. The 500 webpages were requested using the low-end phone. Figure 4 shows the Cumulative Distribution Function (CDF) of the Dom Complete (red curves) and Full Load (blue curves) for both the original pages and JSCleaner pages represented by the solid and the dashed curves, respectively. The figure shows that JSCleaner curves have a relatively lower Dom Complete and Full Load times compared to the original. At the median value (the 50 percentile of the curve), JSCleaner manages to reduce the Full Load by about 38%. In fact, the percentage of reduction increases with higher percentiles, visible by the increased gap between the solid blue and dashed blue curves.

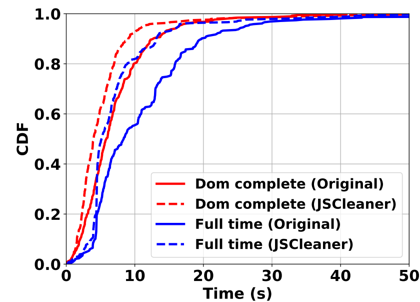


Figure 4: CDF comparison of the dom complete and full load between the original page and JSCleaner page

Figure 5 shows the box plot comparison of different time components starting from the First Meaningful Paint to the Full Load. The light blue boxes represent the original pages, whereas the red boxes represent JSCleaner pages. It can be seen that JSCleaner manages to reduce almost all of the times across all components, in some marginally compared to higher gains in the full load times. Not only visible through the median values but also the lower and higher percentiles. The main gain comes in the dom and full load because thats around the time that most JS elements are evaluated. Figure 6 shows the boxplot results of the total number of requests on the left side of the figure, and the page size on its right side. Results show that JSCleaner has reduced the total number of requests by about 50% compared to the original pages. Similarly, JSCleaner reduces

the page size by about 30% by eliminating a number of non-critical scripts along with their subsequent requests.

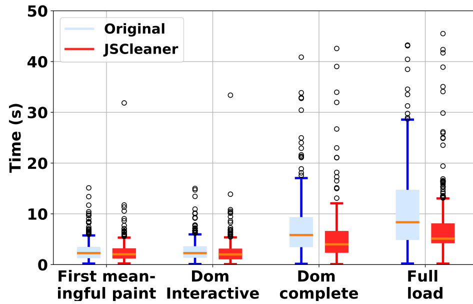


Figure 5: Boxplot comparison of different load times between the original page and JSCleaner page

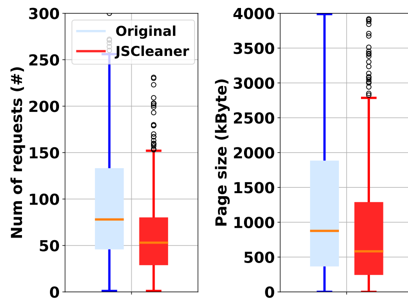


Figure 6: Boxplot of pages' size and total number of requests

6.2.2 Effect of different Cellular Networks. In the previous subsection, we quantified JSCleaner gains purely through the JS processing time reduction. In this subsection, we aim to study if the network conditions play a role in increasing or decreasing the JSCleaner gains seen earlier. The rationale behind this analysis comes from the fact that webpage rendering is often a complex task [24] with numerous dependencies among objects being evaluated, and other objects being requested. Figure 9 shows the CDFs of the Dom Complete and the Full Load times across the three different cellular conditions 2G, 3G and 4G (from left to right). Similar to the CDFs shown earlier when there were no network restrictions, it can be seen that JSCleaner manages to reduce both times, evident by the gap between the dashed and the solid curves. Figure 10 shows the boxplot of the different load times across different network settings. What can be observed from these results is that the JSCleaner gain in reducing the Full Load time is higher in faster networks (4G) compared to slower networks (2G). This can be seen in the summary figures 7 and 8, where the median relative Full Load time reduction is increased from about 21% in 2G to about 27% in 4G. This can be explained by the fact that even though JSCleaner manages to cut down some of the JavaScript processing time, some of this reduction is not fully utilized because of the slow network downloading time of 2G. Despite 4G's higher relative reduction in the Full Load, the absolute reduction in the case of 2G network is 16 seconds, which is quite significant compared to the 8 seconds and 5 seconds reduction of 3G and 4G, respectively.

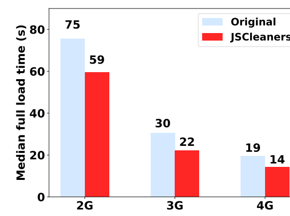


Figure 7: Comparison of the median full load time for different network settings

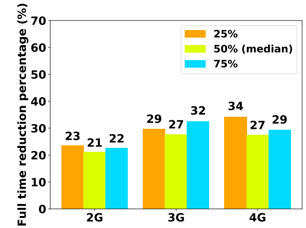


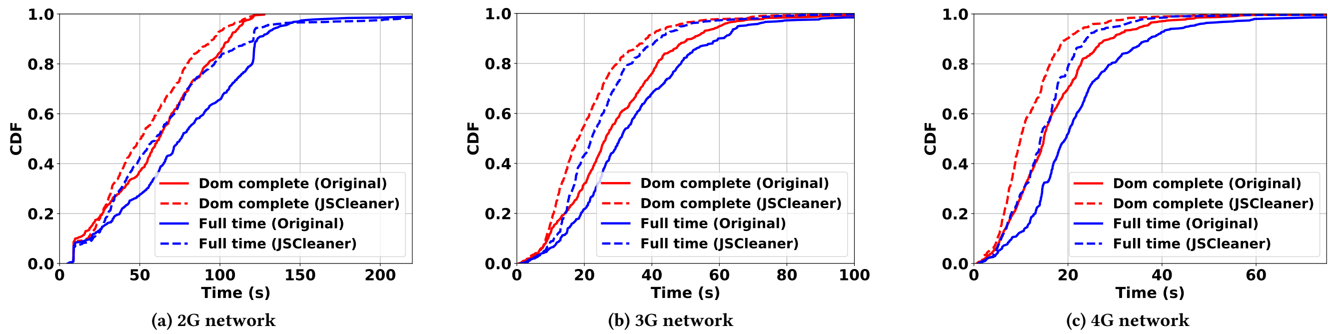
Figure 8: Full load time reductions for different network settings

6.2.3 Effect of High-End Phone. The purpose of this evaluation is to show the pure processing gain of JSCleaner, by testing the effect of using a high-end phone under the emulated 4G network setup, compared to the low-end phone used earlier, under the same 4G settings. We selected Samsung Galaxy S8+ for this evaluation. Results are shown in Figures 11 and 12. Results show that there is about 30% reduction in the median full load time when using JSCleaner, which is slightly higher than the 27% reduction obtained earlier for the low-end phone under the same network settings. It can also be seen that the absolute median full load time has improved compared to the low-end phone results simply because of the better phone resources (3x larger memory and faster CPU). The faster load times are achieved purely due to the faster JS processing capability of the high-end phone.

6.3 JSCleaner Qualitative Evaluation

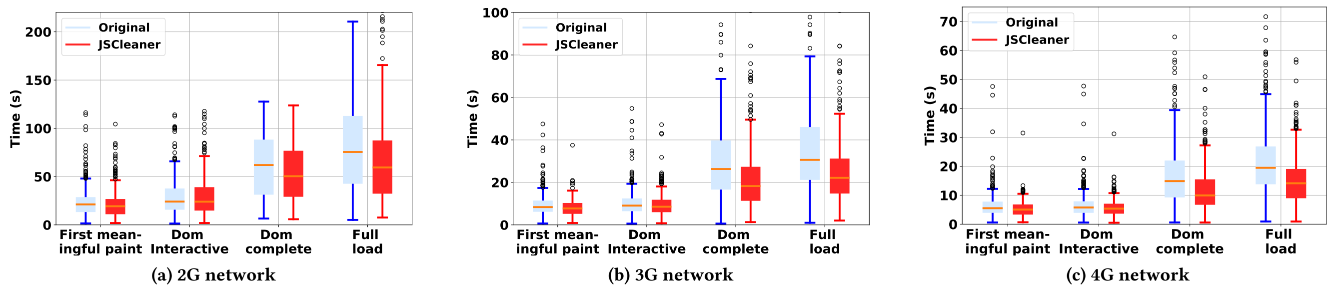
A crucial aspect of evaluating JSCleaner is to analyze the accuracy of JS classification. One of the main goals behind JSCleaner is to simplify the page complexity while retaining the same look and functionality of the original page. We conduct a user evaluation study to compare the visual as well as the functional differences between the original pages and JSCleaner pages for a set of popular websites. For qualitative evaluation of webpages generated by JSCleaner, we randomly selected 10% of the 500 cloned webpages (50 pages). We excluded non-English pages (because of our evaluator pool), as well as pages that were not simplified by JSCleaner (where exactly the same number of JS elements is detected in both the original and the JSCleaner page).

6.3.1 Users Recruitment. Users were recruited from an international university campus by posting online ads on popular social media groups that the students are part of. They were given the chance to reserve a 30 minutes slot from three possible dates. We received more than 120 reservations but ended up with about 103 that actively took part in this study. Recruited students were not part of this work by any means beforehand. They all spoke English and came from different backgrounds and study disciplines, as well as different stages of their university journey (freshmen, juniors, and seniors). The user study was conducted in our lab with three parallel slots, each totaling 35 minutes. During the first five minutes of the slot, we explained the general purpose of the study as well as how to use the evaluation tool. Recruited students were informed that a set of webpages were simplified for performance



(a) 2G network (b) 3G network (c) 4G network

Figure 9: CDF of webpages loading timings comparison (low-end device)



(a) 2G network (b) 3G network (c) 4G network

Figure 10: Boxplot of webpages loading timings comparison (low-end device)

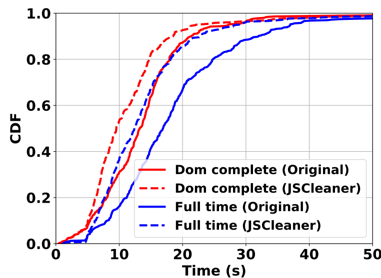


Figure 11: CDF of dom complete & full load for high-end phone with 4G network

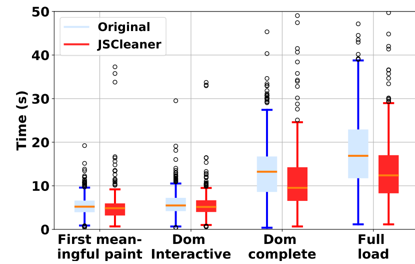


Figure 12: Boxplot of different load times for high-end phone with 4G network

improvement (mainly in terms of load time), and their role is to evaluate the quality of the simplification from different perspectives. In addition, we gave them a practical step by step example of how to do the evaluation. They were then asked to evaluate as many pages as they could within the rest 30-minute slot, and they were compensated with cash for their time. An institutional review board (IRB) approval was given to conduct the user study, and all the team members have completed the required research ethics and compliance training, and were CITI [1] certified.

6.3.2 Evaluation Tool and Metrics. We designed an evaluation tool which consists of a form that is connected to two Firefox browser windows side-by-side, where the left window connects to the MITM proxy that servers the original pages, and the right windows connects to the proxy that serves JSCleaner pages. The evaluation tool randomly selects a URL from the set of 50 pages. It displays the original page of the selected URL on the left side browser window, and the JSCleaner page on the right side window. A sample case for evaluation is shown in Figure 13). The user is asked to compare the two pages and fill in the form with the following considerations:

(1) Look Similarity: where the user is required to rate the look of JSCleaner page compared to the original page using a slider with a 0-to-10 scale. The rate of 0 is interpreted as the two pages do not look similar at all, whereas a rate of 10 means that the two pages look exactly the same. A rate between 0

and 10 refers to the partial similarity that matches the given rate.

(2) Content Completeness: where the user is asked to rate if JSCleaner page maintained all the user relevant content in comparison to the original, regardless of page beautifications or non-relevant content such as advertisements.
(3) Missing Content: which measures if JSCleaner page missed any content such as text, images, ads, videos, layout components, and other embeds (such as maps and tweets).
(4) Functional Similarity: which refers to the similarity of JSCleaner page compared to the original from the functional perspective. The evaluators were asked to count the number of functional elements within the original page (menus, navigational elements, search bars, and image or gallery scroller), and then quantify if the same elements exist in JSCleaner page and whether they are still functional. These elements were identified throughout our dataset observation as the most appearing functional elements, which provide interactivity at the user level.

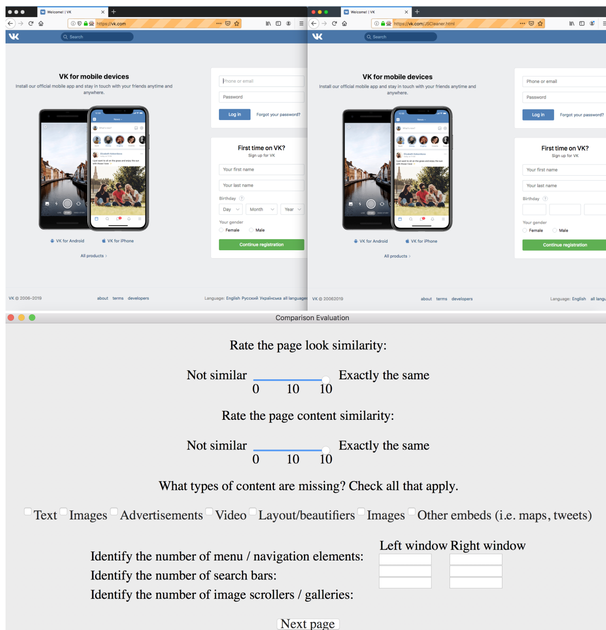


Figure 13: Evaluation tool: A sample case to evaluate

6.3.3 Evaluation Results. The user study was conducted by 103 students, with a total of about 1300 evaluations. Each of the 50 pages was evaluated for at least 20 times, with some pages being evaluated about 30 times due to the random selection process of the evaluation tool, and the students' speed in analyzing the pages.

Figure 14 shows the results of the user study for the page similarity and the content completeness. It can be seen from the figure that according to the evaluators, JSCleaner pages maintained the same look and content to the original pages with a median value of over 93% and 95%, respectively. Results also show a very small 95% confidence interval. Figure 15 illustrates the missing content

Table 3: Functional similarity across different elements

Functional elements	Pages with these elements	Pages with working elements
Menus/Navigation	29	26
Search bars	30	28
Image/gallery scrollers	8	7

breakdown across the different types. The light blue bar shows the percentage of pages that did not have any missing content, whereas the red bar shows the percentage of pages that had some missing elements. It can be seen that for about 90% of the pages, JSCleaner managed to maintain all of the original page elements, apart from some minor layout misalignments that went to about 19%. It's worth noting that from the user perspective, there was no loss in any vital content for more than 90% of the pages. Nevertheless, even for the remaining 10% of the pages, the loss was not significant, evident by the over 95% of content completeness shown earlier in Figure 14. Table 3 shows the number of webpages with the different functional elements, as well as the number of simplified webpages with the maintained functionality. Out of the 50 evaluated pages, we had: 29 pages with active menus and navigation elements, 30 pages with search bars, and 8 pages with active image/gallery scrollers. Results show that the functional elements were fully maintained in about 88% of the pages. For the remaining 12%, some of the navigational elements were not fully functional. However, the lost functionality is not affecting the overall user experience since users reported a much higher content/page similarity to the original page, and none of the core features were lost across any of the pages.

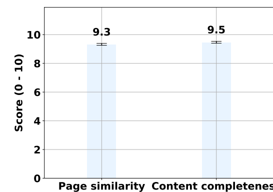


Figure 14: Page similarity and content completeness

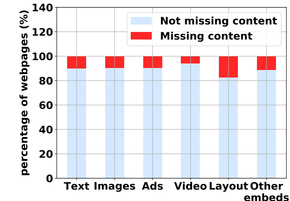


Figure 15: Percentage of pages missing content

7 CONCLUSION

In this paper, we presented JSCleaner, a de-cluttering engine which relies on a rule-based JavaScript classification algorithm for making simplification decisions on webpages. Another intuitive classification approach is to utilize machine learning, by training an appropriate model using sample data, and then utilize it for class prediction. However, the main challenge in this approach is that there exists no known annotated data set for classifying JS. Attempting to create such a set is a challenging task, as it involves JS experts capable of understanding the effect of each script within a page. Another approach is to design a complex human-based annotation platform, where a crowd-sourced environment can be utilized to inspect and annotate JavaScript.

REFERENCES

[1] 2019. CITI Program - Collaborative Institutional Training Initiative. www.citiprogram.org. Accessed: 2019-10-10.

[2] 2019. How we built Facebook Lite for every Android phone and network. https://code.fb.com/android/how-we-built-facebook-lite-for-every-android-phone-and-network/. Accessed: 2019-06-25.

[3] 2019. HTTP Archive. https://httparchive.org/. Accessed: 2019-09-10.

[4] 2019. Majestic Million - Majestic. https://majestic.com/reports/majestic-million. Accessed: 2019-09-10.

[5] 2019. Selenium WebDriver. Browser Automation. https://www.seleniumhq.org/projects/webdriver/. Accessed: 2019-05-14.

[6] 2019. WebPageTest - Website Performance and Optimization Test. https://www.webpagetest.org/. Accessed: 2019-09-10.

[7] Michael Butkiewicz, Harsha V. Madhyastha, and Vyas Sekar. 2011. Understanding Website Complexity: Measurements, Metrics, and Implications. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC '11)*. ACM, New York, NY, USA, 313–328. https://doi.org/10.1145/2068816.2068846

[8] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. 2018. JSExplain: A Double Debugger for JavaScript. In *Companion Proceedings of the The Web Conference 2018*. International World Wide Web Conferences Steering Committee, 691–699.

[9] maximilianhils cortesi and raumfresser. 2019. mitmproxy - an interactive HTTPS proxy. https://mitmproxy.org/. Accessed: 2019-10-13.

[10] Yehia Elkhatib, Gareth Tyson, and Michael Welzl. 2014. Can SPDY really make the web faster?. In *Networking Conference, 2014 IFIP*. IEEE, 1–9.

[11] Mohammad Ghasemisharif, Peter Snyder, Andrius Aucinas, and Benjamin Livshits. 2018. SpeedReader: Reader Mode Made Fast and Private. *CoRR* abs/1811.03661 (2018). arXiv:1811.03661 http://arxiv.org/abs/1811.03661

[12] Google. 2017. Find Out How You Stack Up to New Industry Benchmarks for Mobile Page Speed. https://think.storage.googleapis.com/docs/mobile-page-speed-new-industry-benchmarks.pdf. Accessed: 2019-05-11.

[13] Google. 2019. AMP is a web component framework to easily create user-first web experiences - amp.dev. https://amp.dev. Accessed: 2019-05-05.

[14] Ecma International. 2019. ECMAScript® 2018 Language Specification. http://www.ecma-international.org/ecma-262/9.0/index.html. Accessed: 2019-05-05.

[15] Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. An Operational Semantics for JavaScript. In *Programming Languages and Systems*, G. Ramalingam (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 307–325.

[16] Mozilla and individual contributors. 2005. Document Object Model (DOM).

[17] Mozilla and individual contributors. 2005. The HTML DOM API.

[18] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/netravali

[19] Ravi Netravali, Vikram Nathan, James Mickens, and Hari Balakrishnan. 2018. Vesper: Measuring Time-to-Interactivity for Web Pages. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 217–231. https://www.usenix.org/conference/nsdi18/presentation/netravali-vesper

[20] Addy Osmani. 2018. The cost of JavaScript. https://medium.com/@addyosmani/the-cost-of-javascript-in-2018-7d8950fbb5d4. Accessed: 2019-05-05.

[21] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 346–356. https://doi.org/10.1145/2737924.2737991

[22] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2884781.2884829

[23] Srikanth Sundaresan, Nick Feamster, Renata Teixeira, and Nazanin Magharei. 2013. Community Contribution Award – Measuring and Mitigating Web Performance Bottlenecks in Broadband Access Networks. In *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC '13)*. ACM, New York, NY, USA, 213–226. https://doi.org/10.1145/2504730.2504741

[24] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 473–485. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/wang_xiao

[25] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2016. Speeding up Web Page Loads with Shandian. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 109–122. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/wang

[26] Jihwan Yeo, Changhyun Shin, and Soo-Mook Moon. 2019. Snapshot-based Loading Acceleration of Web Apps with Nondeterministic JavaScript Execution. In *The World Wide Web Conference*. ACM, 2215–2224.

[27] Yasir Zaki, Jay Chen, Thomas Pötsch, Talal Ahmad, and Lakshminarayanan Subramanian. 2014. Dissecting Web Latency in Ghana. In *Proc. of the ACM Internet Measurement Conference (IMC)*. Vancouver, BC, Canada.