JSCleaner: De-Cluttering Mobile Webpages Through JavaScript Cleanup

Paper #183, 6 pages

ABSTRACT

A significant fraction of webpages, which are built using standard Web tools and frameworks, suffer from the *web bloat* problem due to excessive and inefficient usage of JavaScript. Based on analyzing popular webpages, we observe that a reasonable fraction of JavaScript in every page are not truly essential for many of the functional and visual features of the page. In this paper, we propose JSCleaner, a JavaScript de-cluttering engine that aims at simplifying webpages without compromising the page content or functionality. JSCleaner uses a classification algorithm that classifies JavaScript into three main categories: non-critical, translatable, and critical scripts. JSCleaner removes the non-critical scripts from a webpage, replaces the translatable scripts with their HTML outcomes, and preserves the critical scripts. We show that JSCleaner achieves $> 3x$ reduction in page load times coupled with a 50% reduction in the requested objects while maintaining about 80% of the page visual and functional similarity for most pages.

1. INTRODUCTION

The past decade has witnessed a significant increase in webpage complexity. A web browser needs to typically process several complex steps to load one page including: (a) downloading $100+$ objects [4]; (b) spawning $30+$ network connections [20, 6]; (c) issuing $20+$ DNS requests [20]; (d) processing many JavaScript[14]; and (e) processing several layers of recursive requests triggered by JavaScript and HTTP redirections [4]. According to a recent study from Google [8], the average Page Load Time (PLT) for an average mobile landing page is 22 seconds. For bandwidth constrained users, this increased page complexity combined with poor connectivity directly contributes to significant page load times ($> 30-60s$) resulting in an non-interactive web experience [23].

Inefficient and excessive use of JavaScript is a key factor for the increased complexity for mobile pages [9]. Osmani [14] quantified the cost of JavaScript in popular news sites to be extremely high, especially for low-end mobile devices. The Google AMP [9] project aimed to rewrite webpages without the use of JavaScript due to their incurred complexity. In this paper, we micro-analyze 100 popular webpages and quantify the types of JavaScript used in them. We have identified that about 38% of the used JavaScript 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 code, without having the end user incurs the extra cost of requesting and processing these scripts.

Inspired be these findings, we designed *JSCleaner*, a JavaScript cleaner that aims at simplifying modern webpages by optimizing the JavaScript usage. In contrast to other state-of-the-art solutions, JSCleaner does not eliminate the use of JavaScript but rather selectively removes or replaces a portion of these scripts. This is achieved by mainly classifying JavaScript into three main categories: non-critical, translatable, and critical JavaScript. The non-critical scripts are those that do not enrich the end user experience. Typical examples of such scripts are the ones responsible for tracking users and providing statistics for content providers.

This paper makes two key contributions:

- JSCleaner is powered by a novel rule-based classification engine that classifies JavaScript based on 60 distinct features. JSCleaner simplifies mobile webpages and enhances the overall user experience without sacrificing the look and feel of the pages.
- We evaluate JSCleaner by utilizing a diverse set of 100 popular webpages taken from [7]. Our evaluation shows that JSCleaner reduces the page load time by an order of magnitude (>3x), and provides about 50% reduction in requested objects. These gains are achieved while maintaining about 80% similarity to the original page.

2. RELATED WORK

Recently, there has been an increased interest within the research community as well as the industry to tackle the page complexity issues of today's web. In SpeedReader [7], the authors focus on enhancing pages that are rich in content (i.e., pages suitable for the reader mode of web browsers). Their tool is implemented within the rendering pipeline and applies document tree translation before the page is rendered. Unfortunately, SpeedReader is not capable of enhancing the performance of non-readable pages (which heavily utilize JavaScript for content generation). Wprof [21] is another

in-browser tool that acts as a profiler to provide an understanding of the key hindering effects behind the page load times (PLT). Wprof builds a dependency graph between the different browser components and how some might end in blocking others from being active. A key point discovered by Wprof was that JavaScript has a considerable impact on the PLT due to its role in blocking HTML parsing. To address the inefficiencies in the page load process, Shandian [22] was proposed. Shandian a tool that restructures the page load process to speed up its PLT. Shandian uses a proxy server to preload the webpage and sends an initial page DOM to the client. Although Shandian restructures the page resources in a way that enables faster PLTs, it keeps the entire page elements without attempting to simplify any of them, such as JavaScript elements which tend to be resource intensive.

Polaris [12] is another recent tool that tracks data flows during the page loading process, which in turn detects additional edges compared to existing dependency trackers. Since these additional edges allow for more accurate fetch schedules, they can assist web browsers in reducing PLT. Our work is inspired by Polaris, where the overall PLT can be enhanced by reducing JavaScript complexity in modern webpages.

In addition to PLT improvements, different solutions have been proposed to improve the web user experience. In [13], the authors defined load times with respect to interactivity. They introduced a tool that rewrites HTML and JavaScript in webpages to discover the interactivity state of these pages. Despite that JavaScript plays a significant role in webpage performance, and although it became one of the most popular programming languages, its practical performance issues are rarely examined. A recent study [19] shows that inefficient APIs usage is the most common cause of JavaScript performance issues.

From an industry standpoint, both Google and Facebook have attempted to tackle today's web complexity through the proposals of using Google AMP [9] and Facebook Lite [2]. AMP takes a bold approach by redefining and redesigning how pages should be written. It provides web developers with a framework that cuts down the use of JavaScript ironically through the use of a Google JavaScript. Although the idea of simplifying the web by cutting down JavaScript is good in spirit, a balance is required since not all used JavaScript is bad. We believe that cleaning the web usage of JavaScript provides a good balance between keeping the webpages simple while maintaining their critical functionalities. Another major difference between our approach and AMP, is that we aim at simplifying what already exist 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 (including 2G) using low-end phones. To save data and size, Unicode symbols are used to represent icons instead of images. Caching and transcoding of images are also employed for further optimization. Obviously, the features of Facebook Lite are certainly based on careful application-specific design considerations. In contrast, our work aims to provide a generalized framework for content simplification in today's Web.

3. JSCLEANER

JSCleaner focuses on JavaScript since it is the most expensive resource sent to mobile browsers. Many modern webpages utilize JavaScript to produce a set of HTML tags and attach them to the DOM. We refer to these scripts as *translatable scripts*, in the sense that their content can be translated into pure HTML. Besides, and since the major objective of JavaScript is to add functionality to webpages, we define a *critical script* as a script that handles one or more user interactivity features and/or events. Any script that is found not to be critical or translatable is considered as non-critical to the user. The rationale behind the design of JSCleaner is to optimize the JavaScript usage by classifying them into elements that can either be preserved, safely eliminated, optimized and/or replaced.

3.1 Dataset Preparation

For dataset preparation, we examined multiple pages that heavily utilize JavaScript. We selected a diverse set of 100 popular webpages classified as non-readable by [7] (which are the pages that rely on JavaScript for content generation). The resultant dataset includes diverse pages such as: news sites, education, sports, entertainment, travel, and government pages. We utilized a proxy server to clone the content of these pages for two main purposes: first, to extract the set of scripts utilized by them, 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 have modified the mitmproxy web proxy [5] to store all HTTP response messages while maintaining a simple dictionary that maps each HTTP request URL to the stored response message. We also extended the proxy implementation with a caching functionality to act as a cache engine that serves the stored objects.

From every webpage, we extract both: Inline and External scripts. 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. Each script is parsed using our own developed web-oriented parser, which looks for a set of predefined features to extract. Extracted features are then fed to the rule-based classifier, which labels each script based on its features. The classification of JavaScript determines the elimination, replacement, or preservation of a script.

3.2 Feature Engineering

The feature engineering process aims to create a set of features to aid the classification algorithm [18]. The challenge here is that there no existing feature store to aid JSCleaner. To design JavaScript features, we refer to the DOM standard [1], ECMA-262 specification [10], and ECMAScript

language binding for DOM HTML definitions [16]. We selected a list of interfaces which are utilized by JavaScript objects to access the HTML DOM via a set of properties and functions. These interfaces are: Document, HTMLDocument, Node, Element, and Attribute. In addition, we highlighted a list of 51 interfaces which extend the Element interface, such as: HTMLHeadElement, HTMLLinkElement, and HTMLDivElement. These interfaces inherit the properties and functions of their parent interfaces. By properties, we refer to object characteristics, whereas by functions, we refer to the methods associated with objects.

Taking the HTMLDocument interface as an example, an object that implements the HTMLDocument interface inherits the properties of the Document interface (its parent interface), in addition to the following example properties: title, domain, URL, body, images, and scripts. Similarly, an object that implements HTMLDocument inherits the functions of the Document interface. From the list of functions, we could categorize two basic types: reading functions, and writing functions. Before attaching elements to HTML, scripts usually utilize reading functions like getElementsByTagName or getElementById. These functions provide the required access to specific DOM subtree, before adding new elements via a writing function such as createElement and createTextNode, or altering existing element through createAttribute for instance. We created a reference of our selected interfaces, properties and functions to aid us in designing an active list of features. The resultant set consists of 60 features, which form the basis of our platform feature store. We group these features into four major categories, as follows:

1. Property Reading Features (PRF): which indicate a document access for property reading. These include property access of HTMLDocument, Element and Node.

2. HTML Writing Features (HWF): which denote a functional document access for HTML writing, including the insertion or removal of elements. This set merges the functions of both: HTMLDocument and Node.

3. User Interactivity Features (UIF): which represent a set of features correlated with user interactions including events, input features and form interactions.

4. Other Features (OF): which include a set of other features that do not belong to user interactivity, document access or HTML writing. Examples are advertising and user tracking features.

Each of our grouped categories is given a weight to aid the classification algorithm. The weight value of a given feature determines its strength to classify JavaScript. Specifically, the UIF features are given the highest weight, whereas the OF features are given the lowest weight value.

3.3 Web-oriented Parser

Existing JavaScript parsers do not serve our objective. We could only find a formal [15] and an operational [11] semantic parsers for JavaScript. In contrast, we aim to capture the JavaScript access to the HTML DOM. To do so, we designed our own JavaScript web-oriented parser. We rely on our four-group features to aid the parser in recognizing the list of properties and functions utilized by a certain script to access the DOM for reading and/or writing.

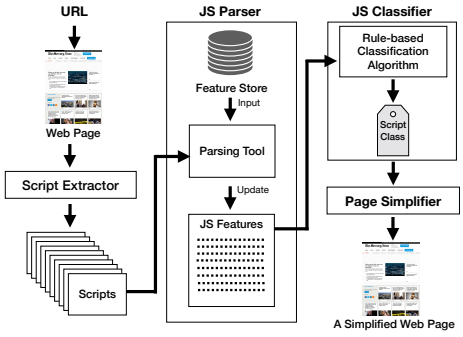

Figure 1: JSCleaner Framework

Our parser utilizes a set of properties and functions stored in a local feature store to generate a tree structure of nodes for each of the parsed scripts. Each tree provides a meaningful representation for a certain script access to DOM, which forms a subset of the original HTML DOM. The tree structure generated after a script parsing process consists of the DOM elements accessed, created, or removed by that script. Our parser generates a list of extracted features for each script to aid the classifier later. Whenever a DOM access is captured in a script, its features list is updated by the parser to reflect that access, as shown in Figure 1.

3.4 Rule-based Classifier

Formally, we define a domain of script documents $D = \{d_1, d_2, ..., d_n\}$. The classification function $f : D \rightarrow C$ assigns a class c_j from the set of predefined classes $C = \{c_1, c_2, ..., c_n\}$ to each document $d_i \in D$. Class assignment is based on a set of predefined logical rules. The classification function considers a list of script features to classify each script according to these rules. Each rule is of type:

$$if \langle DNF formula \rangle \rightarrow \langle class \rangle$$

A DNF (Disjunctive Normal Form) formula is a disjunction of conjunctive clauses [18]. A script is classified under a class if and only if it satisfies the formula of that class, by satisfying at least one of its clauses. The three basic rules we utilized in our classifier are:

$$if \langle \langle E \rangle \vee \langle F \rangle \vee \langle I \rangle \rangle \rightarrow \langle critical \rangle \quad (1)$$

$$
\begin{aligned}
if \langle \langle DF \wedge \neg E \wedge \neg I \wedge \neg F \rangle \\
\vee \langle NF \wedge \neg E \wedge \neg I \wedge \neg F \rangle \\
\vee \langle DP \wedge \neg E \wedge \neg I \wedge \neg F \rangle \\
\vee \langle NP \wedge \neg E \wedge \neg I \wedge \neg F \rangle \rangle \rightarrow \langle trans \rangle
\end{aligned} \tag{2}
$$

$$
if \langle \neg critical \wedge \neg trans \rangle \rightarrow \langle noncritical \rangle \tag{3}
$$

In Table 1, we define the symbols utilized in our rules. Our rule-based classifier starts by assigning a default label to the script being processed. Then, it evaluates the script features to set the final label. Each set of labels belong to a certain class, which is the output of the classification algorithm. For example, the set of labels $L = \{e, i, f\} \subseteq critical$. The label $e \in L$ indicates that a script handles one or more events, whereas labels i and f indicate user input handling and form interactions, respectively. In accordance with the features' weights, UIF features are examined first using eq. 1. The algorithm have access to the feature store to correlate each feature to its corresponding category. If a JavaScript element is found with UIF, it will be classified as $critical$. On the other hand, if no UIF features are found, the algorithm examines the HWF and the PRF features next. Using the rule in eq. 2, it checks if the script being processed tries to access HTML. If any of the HWF features is fetched in a script, it is labeled as w, and a $trans$ class is returned. Similarly, in cases where no HWF features are found, the PRF features are examined, such that if the script is found to have PRF features, it is given the label r. The last category that is being examined by the algorithm is the OF category as in eq. 3. If the previously mentioned features were examined without returning a class for a JavaScript element, the algorithm classifies it as non-critical.

Table 1: Rule-based classification symbols

Symbol	Meaning
e	Event
i	Input
f	Form
DP	Document Property
NP	Node Property
DF	Document Function
NF	Node Function
$critical$	Critical class
$trans$	Translatable class
$noncritical$	Non-critical class

3.5 Simplification Decisions

As shown in Figure 1, for a page to be simplified by JSCleaner, JavaScript elements need to extracted via the extractor module. Every script is then parsed by the web-oriented parser, which outputs a list of features to the rule-based classifier. The classifier, in turn, attaches a class to each script after that. Finally, the page is tackled by the simplifier, which makes a decision on each script based on its class. Every JavaScript 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 translatable, we utilized the HTML output generated by those scripts. To do so, we used Selenium WebDriver [3], along with Firefox 67.0 (64-bit) browser.

For each webpage, JSCleaner prepares an intermediate version which only contains Translatable scripts, and opens it using Selenium to save its source for two reasons: the first is to utilize the Selenium source for replacing the HTML generated by translatable scripts. The second reason is correlated with removing critical scripts from that version. We have experienced loosing some events when utilizing the Selenium source of pages with critical JavaScript. Here, it's worth noting that, in contrast to the original HTML of a page which includes the HTML source along with JavaScript elements, the Selenium source includes the HTML generated by JavaScript (after page rendering) while the original JavaScript elements (both Inline and External) are kept in the rendered page. In translatable JavaScript handling, JSCleaner assumes the following versions of a webpage:

1. $Index_0$: indicates the original HTML of the page that is saved by the proxy when its URL is requested.
2. $Index_1$: refers to the HTML of $Index_0$ after removing both: critical and non-critical JavaScript. Thus, it only includes translatable JavaScript.
3. $Index_2$: refers to the HTML source extracted after opening $Index_1$ using Selenium.
4. $Index_s$: refers to the final version of a webpage generated by JSCleaner after replacing translatable JavaScript found in $Index_2$, and inserting the critical JavaScript found in $Index_0$

To generate $Index_s$ for a webpage, we first use its $Index_0$ to generate the corresponding $Index_1$, which we then open using Selenium to extract the HTML code generated by the replaceable JavaScript. The resultant $Index_2$ 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 JavaScript elements are inserted back before outputting $Index_s$.

4. EVALUATIONS

The evaluation methodology focuses on evaluating the 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 on the end users in terms of page load times and network access. On the other hand, 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 have cloned a set of 100 popular webpages using the mitm proxy, and utilized the same proxy to serve these pages for the purpose of the evaluation. We compared the performance of three versions for each page:

- The Original page
- Semi-simplified page: which is a modified version of the original page, with the removal of non-critical scripts.
- Simplified page: which is the simplified version of the original page, generated by JSCleaner.

4.1 JSCleaner Qualitative Evaluation

The first step towards evaluating JSCleaner is to analyze the accuracy of JavaScript classification. To do so, we compared the visual differences as well as the functionality differences of the original page to the two simplified webpages generated by JSCleaner. One of the main goals behind JSCleaner was to simplify the page complexity while retaining the same look and functionality of the original page.

Visual Comparison: We requested the above described three versions of the 100 webpages using Selenium Webdriver [3] with Firefox 67.0.4, and saved the screen-shots of each version. To assess the webpages visual similarity, we utilized the Structural Similarity Index (SSIM) [24] using Rosebrock [17], where screen-shots of both simplified webpages were compared to the original using OpenCV. The left side of Figure 2 shows the Cumulative Distribution Function (CDF) of the similarity score compared to the original of both versions: Semi-simplified and Simplified. The SSIM is a percentage score out of 100%. The results show that the Semi-simplified version (blue curve) maintains a visual similarity of above 80% to the original for about 25% of webpages. Whereas, the final Simplified version (green curve) achieves the same for only about 50% of the cases. The results confirm that JSCleaner is successful in maintaining a big portion of the original page content. The lower score comes from the cases where JSCleaner classifier has wrongfully categorized certain JavaScript as non-critical. The middle similarity region between 60-90% comes from the fact that JSCleaner does cut down certain content such as advertisements, and certain translatable JavaScript can cause small shift in webpages which will in turn lead to a lower score.

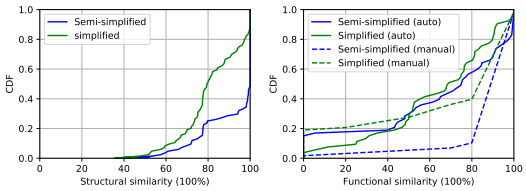

Figure 2: Webpages similarity comparison

Functional Comparison: For the functionality comparison of JSCleaner webpages, we performed two sets of evaluations to assess the performance of JSCleaner. First, we performed a manual comparison, where each component from the original webpage (including buttons, images, search bars, hovering over site navigations, menu buttons, etc.) was compared to its respective counterpart pages and visually analyzed to check if these components retain their full functionality. Second, we created an automatic tool that compares the simplified webpage functionality to the original based on computer vision using OpenCV.

The right side of Figure 2 shows the results of the functionality comparison. For the manual comparison (highlighted by the solid lines), a score of 100% means that the simplified webpage has retained all or almost all functionalities of each component. In contrast, a score of 0% indicated the complete loss of functionality in the webpage. In cases where partial functionalities were being retained, scores in between 0% and 100% were provided depending upon the extent to which webpages were functional. The result shows that about 60% of the simplified pages have retained above 80% functional similarity to the original page (highlighted by the green curve). On the other hand, the semi-simplified pages have retained the same functionality score for more than 90% of the pages.

Now, the dashed lines of the figure shows the functional similarity using the automated testing tool. The tool used a combination of Selenium and jQuery to extract all the tags of a website along with their classes and Ids. Then actions such as hover and click were performed on these tags while capturing screen-shots of their effect. These screen-shots were converted into a binary threshold after the removal of background color and then broken down into components using Opencv.connectedComponents. The broken down components of the semi-simplified and simplified pages were searched against the original image using the integral images. The score was then calculated as: the number of components found divided by the total components detected. The CDF of the automated functionality shows similar trend to the manual functionality results with a slight difference in the middle percentages range, where we see that the results gradually move towards the 100% score. The reason for this difference is that in the manual test the decisions were rigorously more towards maintaining a full functionality or none. Whereas the automatic tool was capable of quantifying the exact percentage of the maintained functionality.

4.2 JSCleaner Quantitative Evaluation

Given that the JSCleaner was successful in simplifying about 60-90% of the investigated webpages in terms of maintaining their visual similarity, we wanted to evaluate the impact of the simplification on the overall user experience. We setup a testing environment using Selenium Webdriver, where a desktop machine requests the three versions of each of the cached webpages for the successfully simplified webpages. We have recorded the HTTP Archive File (HAR) for each of these tests. The HAR file contains the waterfall chart of the network objects requests, in addition to the per object breakdown of the different processes involved in fetching that object. We have parsed these HAR files, where our performance comparison focused on following key metrics: the Page Load Time (PLT), total number of network requests,

and the overall page size. Figure 3 shows the overall PLT comparison between the original webpages and the simplified ones. The left side of the figure shows the CDF of the PLT, whereas the right side of the figure shows the PLT box-plot. The figure shows a significant reduction in the overall PLT, where the simplified page (represented by the green curve) shows a much lower PLT compared to the original. In fact, the median value of the CDF shows more than 3.5x reduction in the overall PLT (about 70%). It can also be seen from the box-plot that the simplified version has a much smaller values for both the 25% and the 75% of the population, where the 75% is even equal to the 50% value of the original webpage. We can also see that the Semi-simplified version also maintains decent reduction in the PLT compared to the original, by only cutting the non-critical JavaScript.

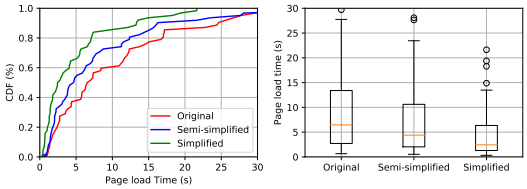

Figure 3: Original vs. simplified PLT CDF and box-plot

Figure 4 shows both the CDF and the box-plot of the page size in MByte. The figure shows very slight reduction in the page size compared to the original page. By contrasting this with the PLT results we have seen earlier, we can conclude that the reduction in the page load times is not as a direct result of reducing the page size, but has more to do with browser evaluation of the non-critical JavaScript (where the browser might end up blocking the rendering pipeline until the successful evaluation of these scripts). This gives a strong signal on the impact of the JavaScript evaluation on the PLT, and showcasing that it is a dominant factor.

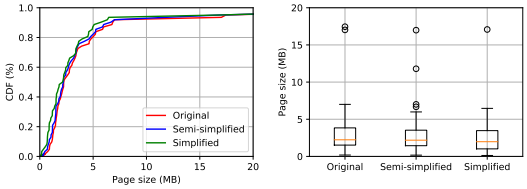

Figure 4: Original vs. simplified page size CDF and box-plot

Finally, figure 5 shows the CDF and the box-plot of the overall object requests issued by the browser to the network. Reducing the overall access to the network for users in developing regions is extremely crucial due to the fact that the network access in these regions is very poor and unreliable. Again the result shows a significant reduction in the overall objects requests due to the fact that we cut a number of the non-critical JavaScript, which tends to cut down a number of subsequent network requests. In fact the simplified webpages cuts down the objects requests by more than half.

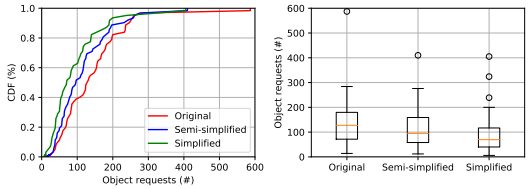

Figure 5: Original vs. simplified total number of objects requests CDF and box-plot

5. DISCUSSION

JSCleaner classifier: JSCleaner relies on a JavaScript classification algorithm as the main bases for it's simplification decisions. In this paper we have implemented a rule-based classification algorithm that classifies JavaScripts based upon 60 distinct features. Another intuitive classification approach is to utilize a machine learning algorithm, 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 exist no known annotated data set for classifying JavaScript. Attempting to create such a set is an extremely challenging task, as it involves using experts capable of understanding the effect of each script within a page.

JavaScript data set annotation: one approach to tackle the creation of an annotated data set is to extend our existing automatic page functionality comparison tool. The tool can automatically classify JavaScript based on the what functionality they achieve. For example, by disabling all scripts apart from one, the tool first makes a visual comparison to decide whether this script is responsible for adding extra content to the page. Then the tool can inspect the page functionality to identify if that script is responsible for that functionality.

JSCleaner challenges: one of the main challenges of JSCleaner is the inaccurate classification of non-critical and translatable scripts. Most of these scripts are complex in nature and it's not intuitive to perfectly design a rule that can generally classify these scripts into their corresponding category.

JSCleaner vs. AMP and Facebook Lite: JSCleaner is one approach that aims at simplifying the complex modern webpages by optimizing JavaScript usage, however there exist a number of recent solutions such as Google AMP that aims at completely re-designing webpages by using a smaller predefined set of HTML elements without the use of JavaScript. Facebook Lite takes another approach at fixing the webpage complexity by having their server fetches data and sends screens to the client as a compressed UI tree to render. They also utilize caching and trans-coding of images to obtain further optimization. Of course the issue with that solution is that its not a general one, since it relies only on enhancing their own app data.

6. REFERENCES

[1] Dom living standard — last updated 15 april 2019. https://dom.spec.whatwg.org. Accessed: 2019-05-05.

[2] 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/, 2019. Accessed: 2019-06-25.

[3] Selenium webdriver. browser automation. https://www.seleniumhq.org/projects/webdriver/, 2019. Accessed: 2019-05-14.

[4] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: Measurements, metrics, and implications. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 313–328, New York, NY, USA, 2011. ACM.

[5] m. cortesi and raumfresser. mitmproxy: a free and open source interactive https proxy. https://mitmproxy.org/, 2019. Accessed: 2019-05-11.

[6] Y. Elkhatib, G. Tyson, and M. Welzl. Can spdy really make the web faster? In *Networking Conference, 2014 IFIP*, pages 1–9. IEEE, 2014.

[7] M. Ghasemisharif, P. Snyder, A. Aucinas, and B. Livshits. Speedreader: Reader mode made fast and private. *CoRR*, abs/1811.03661, 2018.

[8] Google. 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, 2017. Accessed: 2019-05-11.

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

[10] E. International. Ecmascript® 2018 language specification. http://www.ecma-international.org/ecma-262/9.0/index.html. Accessed: 2019-05-05.

[11] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for javascript. In G. Ramalingam, editor, *Programming Languages and Systems*, pages 307–325, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[12] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, 2016. USENIX Association.

[13] R. Netravali, V. Nathan, J. Mickens, and H. Balakrishnan. Vesper: Measuring time-to-interactivity for web pages. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 217–231, Renton, WA, 2018. USENIX Association.

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

[15] D. Park, A. Stefănescu, and G. Roşu. Kjs: A complete formal semantics of javascript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 346–356, New York, NY, USA, 2015. ACM.

[16] W. Recommendation. Appendix d: Ecmascript language binding. https://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/ecma-script-binding.html. Accessed: 2019-05-05.

[17] A. Rosebrock. How-to: Python compare two images. https://www.pyimagesearch.com/2014/09/15/python-compare-two-images/, 2014. Accessed: 2019-06-25.

[18] F. Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, Mar. 2002.

[19] M. Selakovic and M. Pradel. Performance issues and optimizations in javascript: An empirical study. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 61–72, New York, NY, USA, 2016. ACM.

[20] S. Sundaresan, N. Feamster, R. Teixeira, and N. Magharei. 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, pages 213–226, New York, NY, USA, 2013. ACM.

[21] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with wprof. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 473–485, Lombard, IL, 2013. USENIX.

[22] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding up web page loads with shandian. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 109–122, Santa Clara, CA, 2016. USENIX Association.

[23] Y. Zaki, J. Chen, T. Pötsch, T. Ahmad, and L. Subramanian. Dissecting Web Latency in Ghana. In *Proc. of the ACM Internet Measurement Conference (IMC)*, Vancouver, BC, Canada, 2014.

[24] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, April 2004.