To Block or Not to Block: Accelerating Mobile Web Pages On-The-Fly Through JavaScript Classification

MOUMENA CHAQFEH, NYUAD, UAE
MUHAMMAD HASEEB, LUMS, Pakistan
WALEED HASHMI, NYUAD, UAE
PATRICK INSHUTI, NYUAD, UAE
MANESHA RAMESH, NYUAD, UAE
MATTEO VARVELLO, Nokia Bell Labs, USA
LAKSHMI SUBRAMANIAN, NYU, USA
FAREED ZAFFAR, LUMS, Pakistan
YASIR ZAKI, NYUAD, UAE

The increasing complexity of JavaScript (JS) in modern mobile web pages has become a performance bottleneck for low-end mobile phone users, especially in developing regions. In this paper we propose *SlimWeb*, a novel approach that automatically derives lightweight versions of mobile web pages on-the-fly by eliminating non-essential JavaScript that does not impact the core page content and interactive functionality. SlimWeb consists of a JavaScript classification service powered by a supervised Machine Learning (ML) model that provides insights into each JavaScript element embedded in a web page. SlimWeb aims to improve the web browsing experience by predicting the class of each element, such that essential elements are preserved and non-essential elements are blocked by the browsers using the service. We motivate SlimWeb's core design via a preference survey where 306 users overwhelmingly preferred having faster page load times over fetching various categories of non-essential JavaScript. We evaluate SlimWeb across 500 popular web pages in a developing region on real cellular networks, along with a user experience study with 20 real-world users and a usage willingness survey of 588 users. Evaluation results show that SlimWeb achieves 50% reduction in page load time compared to the original pages, and more than 30% reduction compared to competing solutions, while achieving high similarity scores to the original pages measured via a qualitative evaluation study with 62 users. SlimWeb improves the overall user experience metric (defined by Google Lighthouse combining first contentful paint, time to interactive, speed index) by more than 60% compared to the original pages, while maintaining 90-100% of the visual and functional components of most pages.

CCS Concepts: • **Information systems** → **World Wide Web**; **Web interfaces**.

Additional Key Words and Phrases: Mobile Web, JavaScript, Classification, User Experience

Authors' addresses: Moumena Chaqfeh, NYUAD, Abu Dhabi, UAE, moumena@nyu.edu; Muhammad Haseeb, LUMS, Lahore, Pakistan, haseebluminite@gmail.com; Waleed Hashmi, NYUAD, Abu Dhabi, UAE, waleedhashmi@nyu.edu; Patrick Inshuti, NYUAD, Abu Dhabi, UAE, pim219@nyu.edu; Manesha Ramesh, NYUAD, Abu Dhabi, UAE, mr4684@nyu.edu; Matteo Varvello, Nokia Bell Labs, New Jersey, USA, varvello@gmail.com; Lakshmi Subramanian, NYU, New York, USA, lakshmi@cs.nyu.edu; Fareed Zaffar, LUMS, Lahore, Pakistan, fareedz@gmail.com; Yasir Zaki, NYUAD, Abu Dhabi, UAE, yasir.zaki@nyu.edu.

1 INTRODUCTION

JavaScript (JS) has proven to be the most expensive resource to process for web browsers and an essential factor in performance degradation of web pages [63]. The impact of JS is even worse for a large fraction of users who live in low/middle-income countries, where mobile accounts for 87% of the total broadband connections [13], and users solely rely on affordable low-end smartphones to access the web [1]. On these devices, JS processing time has tripled compared to desktops [37], resulting in long delays [63] and poor browsing experience [57, 70]. Nevertheless, the current status of the World Wide Web shows a 45% increase in JS usage by the median mobile page, with only 7% less JS bytes transferred to mobile pages in comparison to desktop (475.1 KB for desktop and 439.9 KB for mobile) [17].

Although the main objective of JS is to provide interactivity to web pages, JS is not always essential to the main page content or interactive functionality [30], and pages can be served with the same original structure and full interactivity while excluding many JS elements. From the user's perspective, if the main contents of the pages are preserved and the functional elements are responding properly, then why should a user with low-end settings tolerate the extra loading time due to a set of analytic elements? From a page developer's perspective, it would be better to satisfy the requirements of more users by offering lighter versions of web pages, instead of losing them due to computationally intensive pages [33]. This paper proposes SlimWeb, a JS classification approach to lighten mobile web pages by blocking non-essential JavaScript. The main challenge that is addressed by SlimWeb is predicting the "criticality" of JS elements without executing their code to block non-essential elements, while maintaining the core visual, interactive, and functional components of mobile pages.

SlimWeb is realized as a service that periodically crawls popular web pages and classifies their embedded JS elements – using a supervised learning model – into *essential* and *non-essential*. SlimWeb shares the classification results with the end-users through a browser plugin that preserves essential JS elements while blocking non-essential ones, effectively producing lighter mobile pages. SlimWeb is primarily designed for low-end smartphones which are the sole means of web access for millions of users in developing regions [7]. Beyond improving Page Load Time (PLT), stripping non-essential JavaScript can also reduce bandwidth and energy consumption on mobile devices, especially given that web browsing is known to cause major drain on the battery lifetime [57, 62].

JavaScript classification is a challenging problem due to the dynamic non-deterministic behavior of its code [67], especially when attempted on-the-fly using low-end mobile phones, since this leaves no room for executing the JS code. Existing research on JS filtering primarily focuses on the identification of advertising and/or tracking JS [42, 47, 49], in addition to malicious JS code snippets [34, 35, 44, 65, 68, 69]. Existing client-side solutions to reduce the cost of JS rely on the integration of a JS blocker as a browser extension [36, 58]. However, these blockers aim to prohibit a specific category of JS (such as Ads) [4, 21, 27], by relying on predefined blocking lists. Although these lists are periodically updated, existing blockers fail to automate the classification of previously-unseen JS elements that can be created from existing libraries by changing the serving domain or obfuscating web pages' code and metadata [20].

With SlimWeb, we aim to address these challenges via a supervised Machine Learning (ML) model. Instead of examining the JS code to infer its semantics and behavior, SlimWeb extracts highly predictive features from the code, and then uses them to classify JS. A labeled dataset is required to train such a JS classifier. Thus, using categories identified by experts of the web community [25], we built a labeled dataset with 127,000 JS elements by scraping more than 20,000 popular pages. Using this dataset, we show that SlimWeb's JS classifier achieves a 90% classification accuracy across the eight categories using 508 carefully selected features. To answer the question of which categories are considered essential or non-essential, we surveyed 306 users from 10 countries in developing regions. Results show that the majority

of users consider *advertising* and *analytics* as non-essential regardless of their network connectivity or mobile phone class. Accordingly, we set advertising and analytics as the default non-essential categories while giving the option to the users to configure their preferences of non-essential JS.

We performed a quantitative evaluation of SlimWeb by installing its plugin in a low-end mobile phone browser, from which 500 popular mobile pages were requested through real cellular networks. We also performed a qualitative evaluation by collecting scores given by 40 users to evaluate SlimWeb when serving 95 popular web pages. Evaluation results show a 50% reduction in the page load time and an improvement of more than 60% in the overall user experience, while maintaining more than 90% of the visual contents and interactive functionality of most pages.

To evaluate SlimWeb with real users, we conducted a user experience study involving 20 users over two weeks. All participants reported that SlimWeb provided faster and lighter pages. Additionally, they gave a median score of 80% in rating the content completeness, as well as retaining the functional elements of the pages, without raising issues related to broken pages. In case of a broken page, SlimWeb plugin allows disabling JS blocking on a per-page basis. To understand the willingness of the users to utilize such a feature for fixing these pages, we surveyed 588 users from different countries, where the majority of them were willing to do so. In summary, the main contributions of this paper are:

- An ML-based JavaScript classifier with 90% accuracy.
- An easy to install plugin that benefits from JavaScript classification in providing lighter versions of web pages for mobile users.
- 50% reduction in page load times compared to the original pages, and around 30% reduction in comparison to existing solutions.
- Around 60% improvement in the user experience, while more than 90% similarity to the original pages is maintained.

2 MOTIVATION

In this section, we motivate the core design principle of SlimWeb, which identifies and removes non-essential JS in web pages to improve user experience without impacting the pages' functionality.

2.1 The Cost of JavaScript

The main browser thread is responsible to run all JS elements in a web page, where long execution times can block the page rendering and delay interactivity [63]. The impact of JS is worsened when web pages are accessed via low-end smartphone devices [57]. Let's consider the cnn.com web page as an example. The Webpagetest tool (a website performance test tool) [66] shows that JS processing requires 2.7 seconds (85.7% of the browser processing time) using a high-end phone (iPhone 8, iOS 12), whereas it requires 39.2 seconds (90.8% of the browser processing time) using a low-end phone (Motorola Moto G4). Given the impact of JS in increasing the load time of web pages and delaying interactivity on low-end smartphones that are the sole means to access the web for millions of users in the developing regions [7], SlimWeb relies on filtering out non-essential JS elements in web pages with the considering the user preferences. Despite the cost of JS, the current status of the World Wide Web shows the utilization of 18 external JS elements on mobile and 19 on desktop at the 50th percentile of web pages. These elements triple the processing time on mobile devices compared to desktops [37]. In locations where connectivity is an issue, the cost of downloading JS resources combined with the poor network connectivity and low-end processing result in a non-interactive experience.

Table 1. Performance of 3 versions of unicef.org.

Metric	Original	JS-Free	Optimized
Number of JS requests	23	0	5
Total page size (MB)	3.8	1.2	2.0
Speed Index (s)	4.8	2.2	3.4
Time to Interactive (s)	15	2.3	6.1

2.2 Non-Essential JavaScript

The design of SlimWeb is inspired by the observation that not *all* JS is essential. We define the non-essential JS as the JS elements with no impact on the page content or interactive functionality. We selected `unicef.org` as an example to show the performance gain of removing non-essential JS, without sacrificing the main content or the functionality of the page. SlimWeb indicates that the page embeds 3 analytics, 4 social, and 1 advertising element. We compared the performance of the *original* version of `unicef.org` with: *JS-Free*, where the page was loaded after disabling JS, and *JS-Optimized*, where the page was loaded with analytics, ads, and social elements blocked, while other JS are preserved.

Table 1 shows the performance results with four metrics. The number of JS requests and the total page size show the difference in JavaScript utilization, and its impact on the total resources' download size, respectively. The effect of JavaScript on the page performance is shown by SpeedIndex [6] and Time to Interactive [8]. As the table shows, removing analytics, advertising, and social elements reduces the number of requests by around 78% (from 23 requests in the original page to 5 requests in the optimized page). This reduction causes a $\sim$50% decrease in the total page size. A considerable impact was also observed in the Time to Interactive metric, which decreased from 15 seconds (original version) to 6.1 seconds (optimized version). The optimized page was visually evaluated in terms of content completeness and functionality. The main content of the page and the functional components were fully preserved; the impact of JS removal has resulted in the removal of an ad element from the top of the page. This example shows the potential of identifying and removing non-essential JS from web pages for the benefit of mobile users without affecting the quality of these pages.

2.3 Removing Non-Essential JS for Faster Pages

To understand the user preferences in removing non-essential JS, we conducted an IRB-approved survey with 306 mobile users from 10 countries in developing regions. The survey was advertised through social media groups, encouraging users with slow connectivity to participate. We asked the participants about the quality of their network connection, and the class of smartphone devices they use to access the web. In addition, we asked them to select all non-essential elements they are willing to remove from web pages for faster browsing including Advertising, Analytics, and Social elements. These elements were listed with easy-to-understand definitions as follows:

- *Advertising*: elements that display ads in the pages you browse.
- *Analytics*: elements that track users' activities by recording their interactions with the page.
- *Social*: elements that enable social features in web pages, such as like and share.

Participants reported the following network connectivity: poor (13%), satisfactory (36.6%), very good (36.9%), excellent (13.3%), and the following mobile phone type: low-end (10%), mid-class (46.4%), and high-end (43.4%). Results show that 99.6% of the participants identified at least one JS category as non-essential and are willing to remove the elements that belong to that category from web pages to achieve better performance. Table 2 shows the percentage of users under different network connectivity conditions who are willing to remove each of the identified JS categories. The table shows

Table 2. Users' preferences in removing Ads/Analytics/Social (split per network condition)

	Excellent	Very Good	Satisfactory	Poor
Ads	92.6%	89.3%	72.7%	82.5%
Analytics	73%	64.6%	50%	57.5%
Social	53.6%	46%	22.7%	22.5%

Table 3. Users' preferences in removing Ads/Analytics/Social (split per phone class)

	High-end	Mid Class	Low-end
Ads	92.4%	80.2%	80.6%
Analytics	70.6%	50%	64.5%
Social	45.8%	33.8%	35.4%

that 92.6% and more than 73% of users under excellent connectivity are willing to remove JS related to advertising and analytics, respectively. It shows a slight decrease in the percentage of users under the other networking conditions who are willing to remove advertising and analytics. Results also show that users under limited connectivity are more interested in social compared to users under high connectivity conditions. On the other hand, users with excellent connectivity are willing to remove more elements from web pages compared to the users under less satisfactory network conditions. Table 3 shows the percentage of users owning different smartphones who considered each of the JS categories non-critical. The table shows similar percentages for each of the categories regardless of the phone specs. It also shows that most of the users are willing to remove ads and analytics JS for faster page load time.

2.4 Limitations of Existing JavaScript Blockers

We investigate the effectiveness of a popular JS blockers (AdBlockPlus [9]) in recognizing non-essential JS code. We analyze 100 popular web pages via Chrome equipped with the AdBlockPlus plugin configured to block ads, analytics, and social elements. Our analysis shows that AdBlockPlus missed about 36% JS files which indeed belong to type ads/analytics/social as per our ground truth analysis (via visual investigation). This result is explained by the fact that existing JS blockers recognize only the URLs (not the content) of the elements to block. In cases where exactly the same code of a given JavaScript element that is known to be hosted by a CDN provider appears in a given page, but with a different URL, existing blockers will not be able to identify it. Motivated by this observation, we aimed to classify JavaScript elements in web pages based on their content, not URLs.

3 RELATED WORK

Reducing the Cost of JavaScript – Web developers often utilize uglifiers [28, 32] to compress JS files before using them in their pages. These uglifiers remove all unnecessary characters (such as white spaces, new lines and comments) from these files without impacting the code functionality. Despite the potential enhancement in JS transmission efficiency due to smaller files, no speedups are expected at the processing level since the browser has to interpret the entire JS anyway. In contrast, JS blocking extensions [4, 21, 27, 36, 46, 58] can reduce the amount of JS transferred while offering potential processing speedups. These blockers rely on updating the blocking lists to prohibit a specific category of JS such as ads or trackers, with a main drawback presented in their inability to predict if an "unknown" JS falls into a target category. This is because the rules in their blocking lists are generated by human annotators [60] which hinders effective

maintenance. For example, the most popular list consists of around 60,000 rules [61] and is not generalized to handle unseen examples. A recent solution called Percival [20] has shown promising results in blocking ads using deep learning. It operates in the browser's rendering pipeline to intercept images obtained during page execution so that it flags potential ads. The evaluation of Percival shows non-negligible performance overhead in both Chromium and Brave [10] browsers on desktops, thus raising some questions on their suitability for (low end) mobile devices.

JavaScript Characterization and Analysis – The state-of-the-art approaches for characterizing and analyzing Web-based JS have been focusing on a particular type of JS, such as tracking [38, 49], and advertising [42, 47]. Web tracking is studied in [52], and analyzed using different types of measurements in [38]. A privacy measurement tool is also proposed to characterize online tracking behaviors. In [49], an ML classifier is proposed to identify tracking JS, while another ML approach is proposed in [47] for detecting advertising and tracking resources. These resources are analyzed in [42], with an evaluation of the effectiveness of ad-blocking blacklists. Another main focus of JS characterization is the insecure web practices [34, 35, 44, 65, 68, 69]. In comparison to the above approaches, we propose to utilize supervised learning to classify each JS element in a web page into the appropriate category, such that elements that fall under the essential categories are preserved, whereas non-essential elements are blocked.

Web Complexity Solutions – In the last decade, we witnessed an increasing interest to tackle the complexity of web pages which has produced fast browsers and in-browser tools. For example with Opera Mini [16], instead of processing JS on the user's device, the processing takes place on a server, which sends the resulting page to the user's browser. This requires the browser to repeatedly communicate with the server whenever a JS function is called. Since most interactions with the page result in calling JS functions, each of these interactions can lead to a delay if the connection between the server and the browser is poor [59]. Brave [10] is another example where predefined domain/URL blacklists are used to block ads for faster pages. Such blacklist-based solutions are unable to detect previously-unseen advertising JS elements.

SpeedReader [40] is a recent feature of Brave which speedups pages suitable for reader mode. SpeedReader is not yet available for mobile phones, and it does not consider pages that utilize JS for content generation. To speedup PLT, Shandian [64] and Polaris [56] restructure the page loading process. Other solutions such as Flywheel [22] and BrowseLite [50] rely on resources' compression. Unlike SlimWeb, these solutions require the browser to download, process, and execute the entire JS of a given page. Google also tackled the complexity of web pages through Accelerated Mobile Pages (AMP) [41], whose performance were studied in [48]. A major difference between SlimWeb and AMP is that the latter provides a framework for developers to create new faster pages, whereas SlimWeb aims to offer lighter versions of pages on-the-fly at the client-end. Because AMP pages can significantly depart from their original versions, both in term of appearance and functionalities, we did not compare SlimWeb's performance to AMP. Our evaluation instead offers, among other things, a comparison with JSCleaner [30], a recent research tool where essential and non-essential JS elements are identified by a rule-based classifier.

4 SLIMWEB

4.1 Architecture

SlimWeb is a solution to speed up page loads and improve the mobile browsing experience. This is achieved via an architecture with two main components: a JavaScript classification service (see Section 4.2) and a browser plugin (see Section 4.3) as shown in Figure 1. The classification service uses a supervised learning classifier to categorize JS elements into one of the following categories: advertising, analytic, social, video, customer success, utility, hosting, and content. These categories are commonly used and defined by web community experts [25]. The first three categories are the

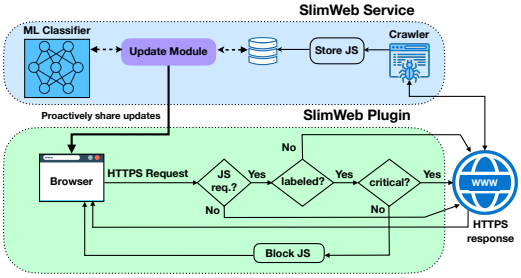


Fig. 1. SlimWeb Architecture

potential *non-essential* categories, while the rest are considered *essential* to the page content or interactivity by default. SlimWeb's JS classification service crawls popular web pages to identify JS elements used in these pages, and then employs the classifier to label these elements and store their categories in a database. It periodically updates and shares the labels with the browser plugin which is responsible for blocking non-essential JS elements when a user visits a web page. These elements are identified based on the labels received from the service. When a web page is requested by the user, the plugin first checks if a label is locally available for each JS element, such that non-essential elements are immediately blocked. In the case of a label absence, the plugin considers the corresponding JS element as essential and requests it from the web.

4.2 JavaScript Classification

SlimWeb employs a novel ML classifier to categorize JS elements embedded in a web page, and then label each of these elements as essential or non-essential according to the user preference. To overcome the challenging task of understanding the non-deterministic behavior of JS [67], the main design goal of SlimWeb is to give an insight into each JS element embedded in a web page, without evaluating the exact behavior of these elements, nor executing their code. The classifier categorizes a given JS element into one of the known JS categories using supervised neural network model, which shows a better and more robust performance over alternatives. Algorithm 1 shows an overview of the classifier. For a given JS element, the classifier outputs a category with a confidence value. If that value is above a predetermined threshold, the predicted category is returned. Otherwise, *unassigned* category is returned. The returned category is used by the plugin to determine if a given JS element is essential or non-essential according to the user preference. For example, if the classifier returns a *social* category for a given JS element while the user setting identifies *social* as non-essential, then the element will be considered non-essential. JS elements with *unassigned* category are conservatively labeled as essential.

Algorithm 1 JavaScript Classification Algorithm

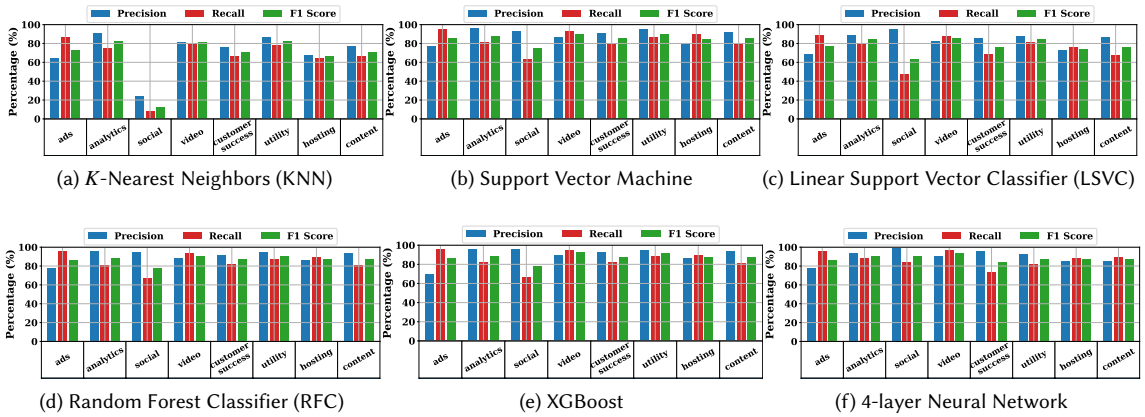
```
1: INPUT threshold, string script[]
2: OUTPUT category
3: procedure ASSIGNCATEGORY(threshold,script)
4:     category, confidence = MLClass(script)
5:     if confidence > threshold then
6:         return category
7:     else
8:         return unassigned
```

Creating a Labeled Dataset To train the supervised learning models, we created a labeled JavaScript dataset by crawling 20,000 popular pages [5] from which we extract their JS elements. For each JS element, we extract its domain for a potential match with data offered by the HTTP Archive repository [45] – which lists existing JS libraries with their associated categories and domains. Our dataset consists of 127,000 JS elements that were labeled with categories out of 500,000 JS elements crawled from 20,000 popular web pages. The category of a given JS element is determined when a match is found in the repository, where the same category of the matching element is used to label the given element. Each JS element is represented by a vector of features and a category label as: $< feature_1, feature_2, feature_n, ..., label >$, where labels are used to determine the criticality of elements and decide whether to block each of them. We used the feature set proposed in [30], which consists of a comprehensive set of features representing all APIs [53, 54] that can be used by JavaScript to interact with the web page's Document Object Model, namely by reading, writing, and event handling. To learn effectively over a small training dataset, we apply Recursive Feature Elimination (RFE)—a feature selection method that removes the less informative features—resulting in 508 features instead of 1,262.

The key benefit of using a classifier to label JS elements instead of matching the repository URLs lies in the ability of the classifier to predict the categories of unknown JS elements embedded in web pages. These elements are hosted in local domains that are unknown to existing blockers (see Section 2.4), and they would not match with any URL in the repository.

Model Training – To select the right ML classifier in SlimWeb, we train an eight class prediction model, which considers eight categories mentioned in [25]. Evaluations are performed on a desktop machine with Intel core i7 CPU @ 3.60GHz x8, 16GB RAM. A simple 4-layer neural network model shows the best performance over the following five supervised learning models: three distance-based classifiers (K-Nearest Neighbors (KNN), Support Vector Machine (SVM) and Linear Support Vector Classifier (LSVC)), and two tree-based classifiers (Random Forest Classifier (RFC) and XG Boost) [31]. The neural network model is built with an input layer, two hidden layers and one output layer. To determine the optimal number of hidden layers, we perform hyper-parameter search optimization, where no significant improvement is observed in the accuracy when using more than two layers. The *ReLU* (Rectified Linear Unit) [23] activation function is used in the hidden layers while the output layers utilize the *softmax* activation function. The total number of neurons in the input layer is equal to the total number of features. Based on hyper-parameter search, the optimal number of neurons is set to 350 in the first layer, and 50 in the second.

Model Evaluation – The learning model is evaluated using *Recall*, *Precision* and *F1-Score* as performance metrics.The detailed evaluation across the different JS categories and ML models is shown in Figure 2. The results of the SVM model (see Figure 2b) shows the lowest recall of 63% in predicting the social category, while the other categories have a recall higher than 79%. Figure 2a presents the results of the KNN model, where a much lower performance is shown in predicting the social category. In contrast, LSVC (Figure 2c) shows a slightly better performance compared to KNN. The results of both tree-based classification models considered in this evaluation (RFC and XG Boost) shown in Figures 2d and 2e, respectively, show similar performance for precision, recall, and f1-score. In general, tree-based classification models provide better performance in comparison to distance-based ones, while the neural network model achieves the best performance (with a slight improvement in precision and recall compared to RFC and XG Boost). Consequently, we select it for JS classifier in SlimWeb.



(a) *K*-Nearest Neighbors (KNN) (b) Support Vector Machine (c) Linear Support Vector Classifier (LSVC)

(d) Random Forest Classifier (RFC) (e) XGBoost (f) 4-layer Neural Network

Fig. 2. ML classification Models overall comparison

4.3 SlimWeb Plugin

SlimWeb leverages a browser plugin[1] to assign each JS element to a class (either essential or non-essential) based on user preference and the JS category. Motivated by the results of the user preference survey presented in Section 2.3, the plugin only considers ads and analytics as non-essential categories by default. Users are able to change SlimWeb's plugin default settings according to their preferences, and to have different configurations on a per-page basis according to their browsing experience.

4.4 Implementation and Deployment

SlimWeb's plugin is implemented using JavaScript for Firefox browser, and is accepted by Mozilla as a Firefox extension. The plugin's local storage is implemented using `IndexedDB` API [15] and allows to store up to 50 MB of JS element labels. The database consists of a sequence of JS entries, where each entry is associated with a URL and assigned label.

SlimWeb is deployed as a server-based solution as described in Section 4.1, where JS elements are classified by a server-side service that regularly updates and shares the classification results with the users' browsers. In addition, we implemented the classification model to run independently on a mobile phone browser using TensorFlow [19] (see Section 5.4 for client-side SlimWeb evaluation). Since the model has to operate at the client side, the content of a given JS file has to be downloaded first by the browser, thus defying the data saving goals for unknown JS. Nevertheless, once the JS element is classified, the class label will be stored within the plugin cache to be used in subsequent visits.

Figure 3 shows an alternative deployment strategy (inside the dotted red area) that allows the plugin to contact SlimWeb's server in the case of JS labels *misses* in the plugin cache (that is, elements whose labels are not part of the updates shared by the server). Such requests can leak private user information, since they allow the server to identify the actual web page being accessed by the user. However, we argue that this privacy concern is not worrisome for the following reasons. First, the case of missing labels is rare since SlimWeb regularly propagates labels of JS elements used in popular pages to avoid extra delays (proportional to the latency between the plugin and the server plus the time needed by the server to fetch and analyze such JS), and to reduce the privacy exposure. Second, JS are commonly shared between pages, e.g. jQuery is used by 73% of the 10 million most popular websites, and websites are encouraged not to host local

[1] https://github.com/comnetsAD/slimweb

versions but rely on CDN-hosted ones. The combination of limited JS uniqueness with SlimWeb's goal to achieve a high cache hit rate for JS labels makes us confident of SlimWeb's limited privacy invasion for this deployment strategy.

5 EVALUATION

This section evaluates SlimWeb with respect to objective metrics like how fast it can load web pages, data savings, and web compatibility. We use two low-end mobile devices: a HUAWEI Y511-U30 (which costs about $89, and is equipped with a dual-core 1.3 GHz Cortex-A7 CPU and 512 MB RAM) and a Xiaomi Remdi Go (which costs about $70, and is equipped with a quad-core 1.4 GHz Cortex-A53 CPU and 1 GB RAM). The Xiaomi phone connects to the Internet over a fast WiFi where network throttling was used to emulate both 3G (downlink/uplink set to 1.6 Mbps/768 Kbps, RTT set to 150ms) and 4G (downlink/uplink set to 12 Mbps, RTT set to 70ms) networks, whereas the HUAWEI phone connects to the Internet using real 3G/4G networks.

Each mobile device connects via USB to a Linux machine which is used to run the following browser automation tools: Browsertime [51] with the Firefox browser and Lighthouse [11] with both Chrome and Brave. Such browser automation tools are used to automate both web page loads and telemetry collection, e.g. performance metrics and network requests. Our testbed further consists of a second Linux machine which acts as SlimWeb's server and runs the JS labeling service (see Figure 1). This machine is located in the same lab as the Xiaomi Remdi Go and about ∼100 ms (median latency) away from the HUAWEI Y511-U30, and is used in all the evaluations apart from the independent phone deployment scenario evaluated in 5.4

In addition to Firefox, we also evaluate SlimWeb when used in conjunction with the more popular Chrome and Brave, an ad-blocking browser with growing popularity [10]. Due to the lack of plugin support for Chrome and Brave in Android, for the evaluation purpose, we implemented SlimWeb client-side capabilities in mitmproxy [12], a man-in-the middle proxy which can intercept (and block) regular HTTP(S) traffic. To operate on encrypted traffic, we install mitmproxy's root certificate authority on the mobile phones. We deploy mitmproxy on another Linux machine located in the same LAN as the mobile phones to minimize latency – which would normally be 0 as SlimWeb operates as a browser plugin. This machine is employed only to extend our experiments to Chrome and Brave, and is not a core component of SlimWeb. All our evaluations are performed over subsets from the 11,000 unique pages collected in [26], which consists of the anonymized browsing history of 82 undergraduate students from Pakistan (where low-end smartphones are common) in a period of 100 days.

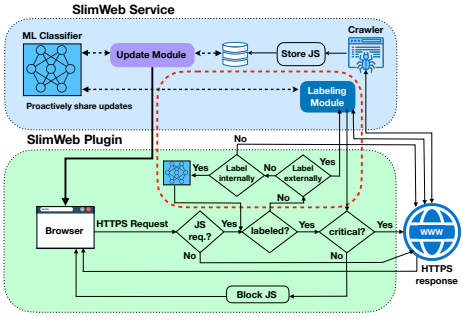


Fig. 3. Alternative SlimWeb deployment option

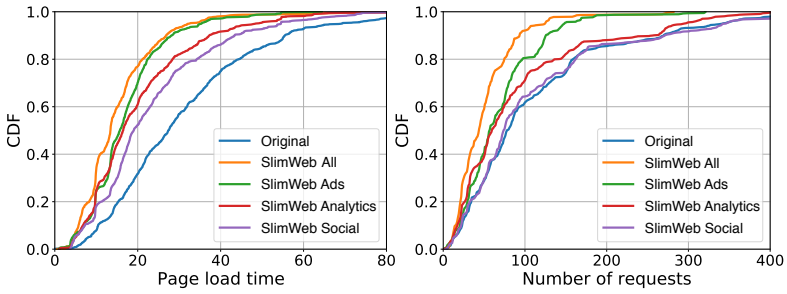


Fig. 4. SlimWeb configurations: Quantitative results

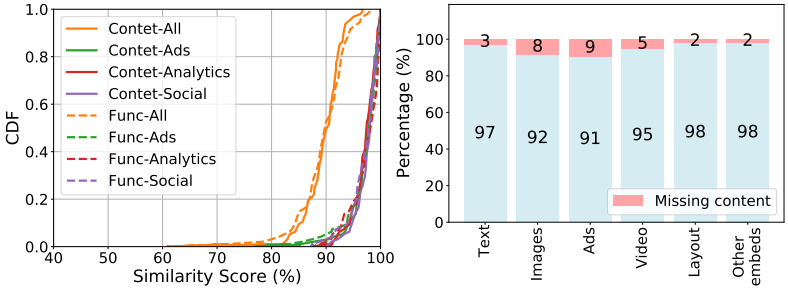


Fig. 5. SlimWeb configurations: Qualitative results

5.1 SlimWeb Configurations Evaluation

The user preference survey indicates that users are willing to block the following JS categories: ads, analytics, and social. To understand the effect of blocking these categories we devised three evaluations, blocking a different category each time, in addition to concurrently blocks all of the three categories. For this experiment, we use Chrome running on the Xiaomi phone over an emulated 3G network. We load (10 times) the 95 most popular pages from the 11,000 unique pages collected in [26].

Figure 4 shows the Cumulative Distribution Functions (CDFs) for page load time, and number of network requests. Figure 4a shows that blocking ads only (green curve) achieves the highest PLT reductions compared to the other two categories, which is almost comparable to blocking all categories (orange curve). The figure also shows that social has the least impact on the PLT compared to the other categories, followed by blocking analytics. Although blocking ads achieves the closest reductions to blocking all categories, it does not come close in terms of reducing the number of network requests (Figure 4b). This suggests that it is beneficial for users in developing regions to block all the three categories to achieve the most data savings, with more than 50% reduction in page size at the median (from 1,300 to 600 KB). Additionally, reducing the number of network requests indirectly improves the phone energy consumption, given that it reduces the usage of the cellular interface.

To evaluate the effect of SlimWeb on the quality of the pages, we conducted a user study with 62 participants who are recruited from an international university, and each was given a random set of 10 pages out of the full set, and is

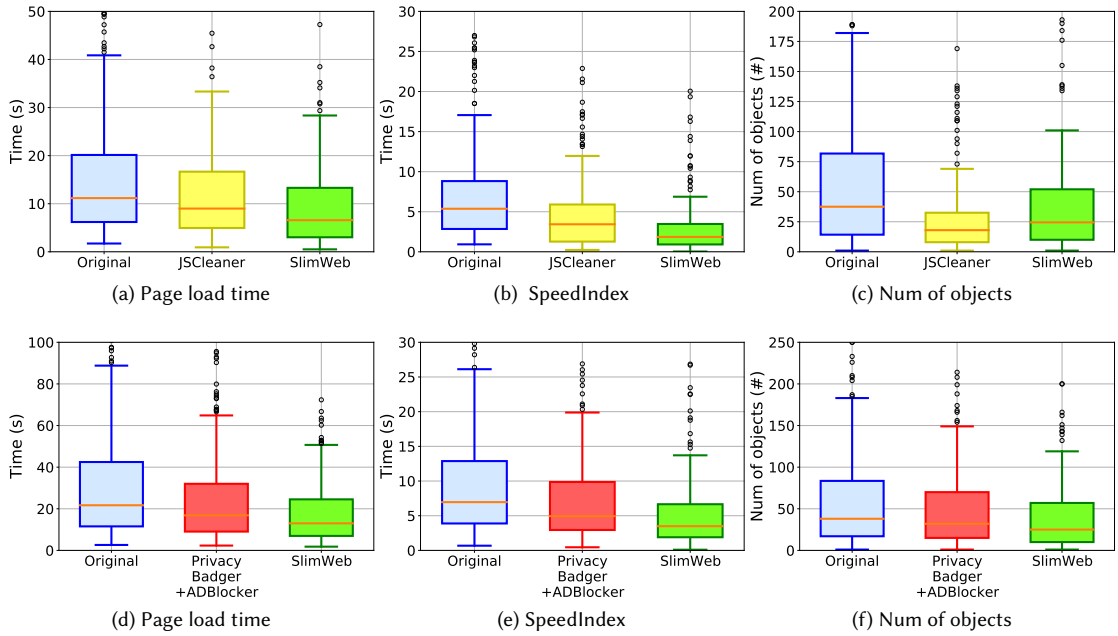
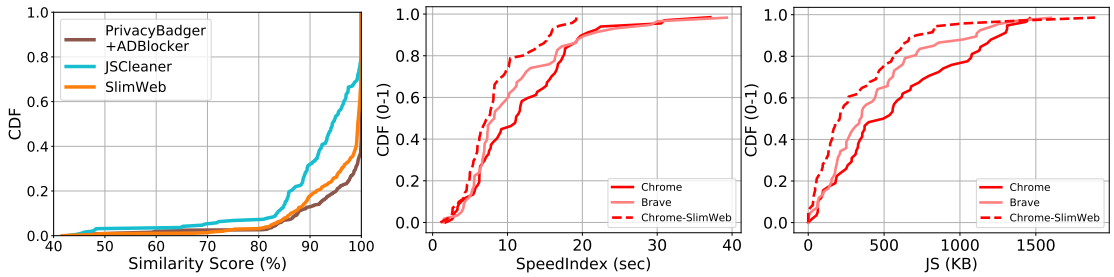


Fig. 6. SlimWeb quantitative evaluation results in comparison to existing solutions

asked to compare the four different versions of the pages to their original counterpart in terms of content similarity and functionality. The users gave scores between 0 (worst) and 10 (best). An institutional review board (IRB) approval was given to conduct the study, and all the team members have completed the required research ethics and compliance training, and are CITI [3] certified. Figure 5a shows the CDFs of the content similarity (the solid lines) and the functional similarity (the dashed lines) to the original pages in a form of a percentage score for each SlimWeb configuration. It can be seen that all individual configurations achieve scores of more than 90% similarity for both content and functionality. While a combination of all the configurations shows a lower similarity score, the overall scores are still above 90% for half of the pages, and the rest of the pages achieve more than 85% similarity. Finally, Figure 5b shows the percentage of pages missing content split across the content types. Users reported 9% of pages having missing ads, followed by 8% with missing images. The rest of the types are only missing in $< 5\%$ of the pages.

5.2 Comparison With Existing Solutions

Here, we compare SlimWeb with several key "competitors": JSCleaner, and Privacy Badger combined with ADBlock. JSCleaner [30] is a recent solution that offer simplified pages via a proxy server (see Section 3). Privacy Badger [39] is a browser extension that blocks both advertising and tracking, whereas ADBlock [21] is one of the most popular ad-blocking browser extension that is used by tens of millions of users worldwide. Our comparison is both *quantitative*, i.e., focusing on speed and data usage, and *qualitative*, i.e., focusing on similarity of the lightened pages with respect to the original pages. We leverage Firefox browser using the HUAWEI Y511-U30 phone equipped with the SlimWeb plugin. Browsertime is used for automation and no proxy is used given that Firefox mobile supports addons—with the exception



(a) CDF of similarity score (b) CDF of SpeedIndex (sec) (c) CDF of JS sizes (KB)

Fig. 7. Comparing SlimWeb (over Chrome) with 1) PrivacyBadger+ADBlocker and JSCleaner ; 2) Chrome and Brave.

of JSCleaner which is a proxy-based solution. We consider real network conditions to evaluate the top 500 most popular pages from [26].

Quantitative Comparison – The following metrics are considered in the quantitative evaluation: a) Page Load Time, b) Number of objects per page, and c) Speed Index (which is an important user experience metric). Figures 6a to 6c show the performance of SlimWeb in comparison to both: the original and the JSCleaner pages over the 4G cellular network. Figure 6a shows that SlimWeb significantly improves the PLT in comparison to the original and the JSCleaner pages. Specifically, SlimWeb reduces the PLT by more than 50% in comparison to the original, and by around 30% in comparison to JSCleaner. This can be observed by comparing the median value of SlimWeb (green box) to the original (blue box) and the JSCleaner (yellow box). Figures 6b and 6c show that SlimWeb reduces the SpeedIndex by more than 60%, and the number of web objects by more than 40% in comparison to the original. Despite SlimWeb preserving more objects in comparison to JSCleaner, it improves the user experience by an additional 30% reduction in the SpeedIndex.

Figures 6d to 6f show the performance of SlimWeb over 3G network in comparison to the original and the privacy-badger+AdBlock. Results show about 50% reduction in the PLT when using SlimWeb (green box), in comparison to a reduction of 25% with privacy-badger+AdBlock (red box). Similar observations can be seen in Figures 6e and 6f, where SlimWeb achieves a significant reduction in the number of objects and SpeedIndex in comparison to the original and the privacy-badger+AdBlock. In conclusion, results presented in Figure 6d to 6f prove that the improvement seen in SlimWeb cannot be achieved by existing blocking solutions, even when using a combination of these solutions together. This is due to the use of supervised learning in SlimWeb to classify JS elements, while other solutions rely on blocking lists that ignore previously-unseen JS.

Qualitative Comparison – For qualitative evaluation, we compute the similarity scores of pages lightened by SlimWeb, JSCleaner, PrivacyBadger+AdBlock with respect to the original pages. The similarity score of each version is computed using *PQual* [43], which is a tool that automates the qualitative evaluation of web pages using computer vision. Figure 7a compares the similarity scores of each page versions as CDFs. The figure shows that more than 80% of the SlimWeb pages have a score that exceeds 90%. Results confirm that SlimWeb does not affect the overall content and the functionality of the pages by blocking the non-essential JS. In contrast, JSCleaner affects the quality for a higher percentage of pages. Results also confirm that there is only a small negligible difference in the similarity scores between the SlimWeb and PrivacyBadger+AdBlock.

5.3 Comparison With Chrome and Brave

In this section, we investigate SlimWeb's performance using the Chrome (version 87.0.4280.101) and Brave (version 1.18.77) browsers, respectively the most popular Android browser [55] and an upcoming privacy-preserving browser with integrated ad-blocking capabilities [10]. Due to the lack of plugin support for both Chrome and Brave in Android, we resort to SlimWeb's proxy for these experiments. Browser automation is realized via Lighthouse and a 4G network is emulated. The same corpus of pages used in Section 5.1 is also used here, and the test device is the Xiaomi phone.

Figure 7 summarizes our results with respect to performance – due to space limitations we only report on SpeedIndex but similar observations hold for other metrics – and JS usage. We report results for both regular Chrome and Brave, i.e., when the browsers are directly connected to the Internet, and for *Chrome-SlimWeb* where Chrome's traffic is intercepted by SlimWeb's proxy. We also experimented with Brave via our SlimWeb proxy; results are omitted since we observed artifacts from the interaction between Brave's adblock and our proxy. Figure 7b shows significant acceleration offered by Chrome-SlimWeb compared to both regular Chrome and Brave. For example, the median SpeedIndex reduces by 4 and 1 seconds when comparing Chrome-SlimWeb with Chrome and Brave, respectively. The lower reduction observed when comparing with Brave is due to its native ad-blocker which already reduces web page complexity offering significant speedups compared to Chrome [2].

To verify the latter observation, Figure 7c shows the CDFs of the bytes of JS files transferred during the experiments. Brave serves overall lighter pages compared with Chrome, e.g. median JS size reduces from 446 to 337KB ($\sim$25% reduction). However, SlimWeb manages to offer additional 25% savings, e.g., reducing the median JS size down to only 204KB. These benefits come at the expense of marginal loss of content/functional similarity to the original pages (10-15% reduction shown earlier in Figure 5a). In Section 5.5, we demonstrate that many users are willing to use SlimWeb even with this marginal loss. By downloading only essential JS, SlimWeb achieves better PLT in comparison to both Chrome and Brave.

5.4 Evaluation of Client-Side SlimWeb Deployment

Here, we evaluate the alternative deployment scenario described in Section 4.4, where SlimWeb is implemented in a form of an independent browser plugin that runs the classification model on the phone without communicating to a server. The evaluation is performed over two different sets of live pages taken from two popular page ranking datasets, namely Tranco [18] (which provides the top visited landing pages from Alexa, Cisco Umbrella, and Majestic), and Hispar [24] (which provides the top visited internal pages). We chose these two sources due to the fact that they are the state-of-the-art page popularity listings, in addition to our aim to evaluate the client-side classification over an online setting from the original content servers rather than evaluating our previously cached pages. We evaluated a total of 500 pages with the top 250 sites from each list. Each page was requested three independent times, first without the client-side plugin, and then with the plugin.

Figure 8 shows the delta improvements over the original pages without the plugin for the three different metrics: DOM-interactive, speedIndex, and the page load time. The figure shows that across all the three metrics, the plugin achieves a positive delta improvement for about 80% of the pages. In the case of the page load time, the median delta improvement is about 2.1 seconds (which is about 33% reduction from the median page load time which is about 6.25 seconds). This is a significant improvement that does not require any infrastructural support to deploy SlimWeb, given that the mobile phone can dynamically label and block non-essential JS requests on-the-fly with an independent client-side plugin.

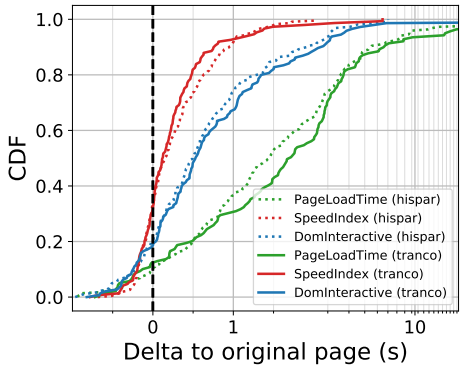


Fig. 8. SlimWeb's client-side deployment evaluation results

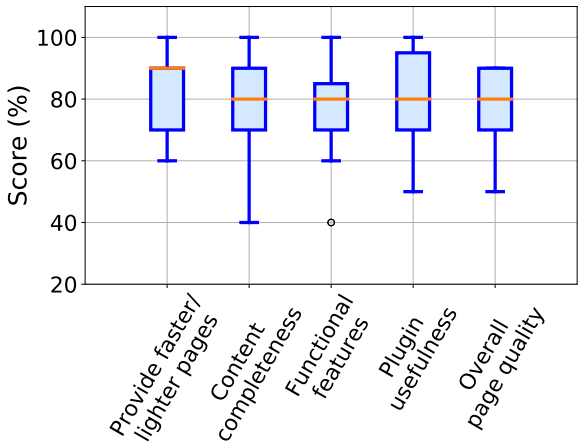


Fig. 9. User scores

5.5 Evaluation with Real Users

Real users play a fundamental role in SlimWeb's success. Therefore, it is important to evaluate it with real users and assess their willingness to help correct potential web compatibility issues caused by SlimWeb. Our evaluation with real users include a user study encompassing 20 participants who installed and run SlimWeb's plugin on Firefox over 14 days, and a survey covering 588 volunteers. We evaluated SlimWeb by requesting 20 users to install the plugin on their smartphones and browse the web from these devices for a period of two weeks. Participants were then requested to respond to a set of survey questions by the end of the evaluation period. Out of 20 participants, 19 responded to the survey questions. All participants believe that SlimWeb is useful in providing lighter versions of web pages. Results show that the plugin helps 63% of the users to browse more pages with their data plan. Around 68% of the users agree that the feedback on broken pages helps in improving the performance of the plugin. The users were asked to rate the following on a 0-10 scale [29]: the usefulness of the plugin, the content completeness, the functional and interactivity features, the performance and the quality of the lightened pages. Figure 9 depicts a summary of the users' responses in a percentage score, where a median of at least 80% for all of the requested assessments is shown.

The second survey results show that most users are willing to use SlimWeb despite the probability of breaking pages. Most users are even willing to put in the effort to fix the broken pages. Participants to the survey are asked to indicate their connectivity condition and phone type. Further, they are introduced to the concept of the SlimWeb's plugin as a solution to speed up web page loads and improve their browsing experience, which might come at the expense of removing certain components from the pages, and/or breaking some of them. After that, the users are asked to answer two questions. The first question asks if the user is willing to use the plugin assuming that it might break around 15 to 20% of the pages (some pages might be missing certain components, and in some off-cases the pages might be broken). Two examples of broken pages are shown to clarify how broken pages might look like (in each example, the original page is shown side by side with the lightened broken page). The second question asks if the user is willing to put in the effort to fix a broken page (assuming that the plugin gives the option to bring some of the elements back to the page, by listing all of the blocked elements via an interface and allowing to disable the blocking of each of them to test the page afterwards).

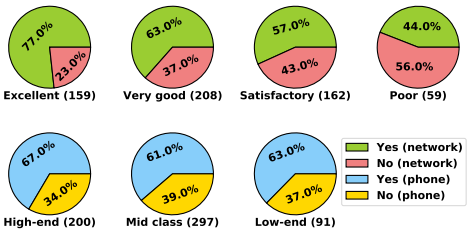


Fig. 10. Willingness to use SlimWeb, split as a function of network type (top), and phone class (bottom).

Fig. 11. Willingness to fix broken pages, split as a function of network (top), and phone class (bottom).

We first utilized google surveys [14] as a paid platform to recruit users across 4 different developing countries. Additionally, we posted the survey on multiple social media groups to reach out to users in countries that are not available for selection in google surveys. We offered the survey in three languages (Arabic, English and Spanish). A total of 588 participants responded to the survey from multiple developing countries including: Philippines, Ghana, South Africa, Pakistan, Colombia, Kenya, Nigeria, Egypt, Yemen, Syria, and Iraq. Responses to the first and the second question are summarized in Figures 10 and 11, respectively (split per network type in the first row of each of the figures, and per phone class in the second row). As Figure 10 shows, most users are willing to use SlimWeb despite the probability of breaking pages, were the highest percentages are shown with excellent connectivity and high-end phones. Most users are also willing to put in the effort to fix the broken pages as Figure 11 shows, where poorly connected users present the highest percentage.

6 CONCLUSION

This paper proposes and evaluates SlimWeb, a novel ML-driven approach that automatically derives lightweight versions of web pages by eliminating non-essential JavaScript. SlimWeb aims to improve the web browsing experience of users with low-end devices and poor connectivity by predicting the class of each JS element, such that essential elements are preserved while non-essential elements are blocked. Evaluation results across 500 web pages and both 3G and 4G networks show that SlimWeb outperforms existing solutions while providing high similarity to the original web pages. A user study with a group of 20 users shows the practical utility of SlimWeb for mobile users in developing regions. In our future work, we aim to perform a large-scale roll-out of SlimWeb.

REFERENCES

[1] 2019. The age of digital interdependence. https://www.un.org/en/pdfs/DigitalCooperation-report-for%20web.pdf. Accessed: 2021-06-16.

[2] 2019. Brave 1.0 Performance: Methodology and Results. https://brave.com/brave-one-dot-zero-performance-methodology-and-results/. Accessed: 2021-01-12.

[3] 2019. CITI Program - Collaborative Institutional Training Initiative. www.citiprogram.org. Accessed: 2019-10-10.

[4] 2019. The Fastest, Most-Powerful Ad Blocker. https://ublock.org/. Accessed: 2020-01-6.

[5] 2019. The Majestic Million. https://majestic.com/reports/majestic-million. Accessed: 2019-11-11.

[6] 2019. Speed Index. https://web.dev/speed-index/. Accessed: 2021-01-12.

[7] 2019. The State of Mobile Internet Connectivity 2019. https://www.gsma.com/mobilefordevelopment/wp-content/uploads/2019/07/GSMA-State-of-Mobile-Internet-Connectivity-Report-2019.pdf. Accessed: 2020-03-17.

[8] 2019. Time to Interactive. https://web.dev/interactive/. Accessed: 2021-01-12.

[9] 2020. Adblock Plus | The world #1 free ad blocker. https://adblockplus.org/. Accessed: 2021-06-28.

[10] 2020. Brave: the Privacy Preserving Browser. https://brave.com/. Accessed: 2021-01-12.

[11] 2020. Lighthouse. https://developers.google.com/web/tools/lighthouse. Accessed: 2021-01-12.

[12] 2020. Mitmproxy – a Free and Open Source Interactive HTTPS proxy. https://mitmproxy.org/. Accessed: 2021-01-12.

[13] 2020. The State of Mobile Internet Connectivity 2020. https://www.gsma.com/r/wp-content/uploads/2020/09/GSMA-State-of-Mobile-Internet-Connectivity-Report-2020.pdf. Accessed: 2021-06-16.

[14] 2021. Google Surveys. https://surveys.google.com/. Accessed: 2021-01-11.

[15] 2021. IndexedDB API. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API. Accessed: 2021-10-08.

[16] 2021. Opera Mini for Android. https://www.opera.com/mobile/mini. Accessed: 2021-01-11.

[17] 2021. Report: State of JavaScript. https://httparchive.org/reports/state-of-javascript#bytesJs. Accessed: 2021-05-1.

[18] 2021. A Research-Oriented Top Sites Ranking Hardened Against Manipulation. Retrieved June 6, 2021 from https://tranco-list.eu/

[19] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[20] Zainul Abi Din, Panagiotis Tigas, Samuel T King, and Benjamin Livshits. 2020. PERCIVAL: Making in-browser perceptual ad blocking practical with deep learning. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 387–400.

[21] AdBlock. 2009. Surf the web without annoying pop ups and ads. https://getadblock.com/. Accessed: 2020-05-02.

[22] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. 2015. Flywheel: Google's data compression proxy for the mobile web. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 367–380.

[23] Abien Fred Agarap. 2018. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375* (2018).

[24] Waqar Aqeel, Balakrishnan Chandrasekaran, Anja Feldmann, and Bruce M Maggs. 2020. On Landing and Internal Web Pages: The Strange Case of Jekyll and Hyde in Web Performance Measurement. In *Proceedings of the ACM Internet Measurement Conference*. 680–695.

[25] HTTP Archive. 2019. Third Parties | 2019 | The Web Almanac by HTTP Archive. https://almanac.httparchive.org/en/2019/third-parties. Accessed: 2020-01-2.

[26] Muhammad Ahmad Bashir, Umar Farooq, Maryam Shahid, Muhammad Fareed Zaffar, and Christo Wilson. 2019. Quantity vs. Quality: Evaluating User Interest Profiles Using Ad Preference Managers. In *NDSS*.

[27] Mark Bauman and Ray Bonander. 2017. Advertisement blocker circumvention system. US Patent App. 15/166,217.

[28] Mihai Bazon. 2012. UglifyJS. http://lisperator.net/uglifyjs/. Accessed: 2020-05-01.

[29] James Carifio and Rocco J Perla. 2007. Ten common misunderstandings, misconceptions, persistent myths and urban legends about Likert scales and Likert response formats and their antidotes. *Journal of social sciences* 3, 3 (2007), 106–116.

[30] 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*. 763–773.

[31] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[32] CircleCell. 2011. JSCompress - The JavaScript Compression Tool. https://jscompress.com/. Accessed: 2020-05-01.

[33] Mary Ellen Coe. 2019. Milliseconds earn millions: Why mobile speed can slow or grow your business. https://www-thinkwithgoogle-com.cdn.ampproject.org/c/s/www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-site-speed-importance/amp/. Accessed: 2020-09-15.

[34] Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2010. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the 19th international conference on World wide web*. ACM, Association for Computing Machinery, New York, NY, United States, Raleigh North Carolina USA, 281–290.

[35] Charlie Curtsinger, Benjamin Livshits, Benjamin G Zorn, and Christian Seifert. 2011. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection.. In *USENIX Security Symposium*. San Francisco, USENIX Association, USA, 33–48.

[36] Sybu Data. 2016. Sybu JavaScript Blocker – Google Chrome Extension. https://sybu.co.za/wp/projects/js-blocker/. Accessed: 2020-05-02.

[37] Houssein Djirdeh. 2019. JavaScript | 2019 | The Web Almanac by HTTP Archive. https://almanac.httparchive.org/en/2019/javascript. Accessed: 2020-01-2.

[38] Steven Englehardt and Arvind Narayanan. 2016. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, Springer Publishing Company, Incorporated, Vienna Austria, 1388–1401.

[39] Electronic Frontier Foundation. 2020. Privacy Badger. https://privacybadger.org/. Accessed: 2020-09-14.

[40] 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

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

[42] Saad Sajid Hashmi, Muhammad Ikram, and Mohamed Ali Kaafar. 2019. A Longitudinal Analysis of Online Ad-Blocking Blacklists. arXiv:1906.00166 [cs.CR]

[43] Waleed Hashmi, Moumena Chaqfeh, Lakshmi Subramanian, and Yasir Zaki. 2020. PQual: Automating Web Pages Qualitative Evaluation. In *The Adjunct Publication of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual (previously Minneapolis, Minnesota, USA)) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA.

[44] Xincheng He, Lei Xu, and Chunliu Cha. 2018. Malicious JavaScript Code Detection Based on Hybrid Analysis. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, IEEE, Nara, Japan, 365–374.

[45] Patrick Hulce. 2019. third-party-web/entities.json5 at 8afa2d8cadddec8f0db39e7d715c07e85fb0f8ec · patrickhulce/third-party-web. https://github.com/patrickhulce/third-party-web/blob/8afa2d8cadddec8f0db39e7d715c07e85fb0f8ec/data/entities.json5. Accessed: 2020-01-2.

[46] Cliqz International. 2009. Ghostery Makes the Web Cleaner, Faster and Safer. https://www.ghostery.com/. Accessed: 2020-05-2.

[47] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. 2018. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. arXiv:1805.09155 [cs.CY]

[48] Byungjin Jun, Fabián E Bustamante, Sung Yoon Whang, and Zachary S Bischof. 2019. AMP up your Mobile Web Experience: Characterizing the Impact of Google's Accelerated Mobile Project. In *The 25th Annual International Conference on Mobile Computing and Networking*. 1–14.

[49] Andrew J. Kaizer and Minaxi Gupta. 2016. Towards Automatic Identification of JavaScript-Oriented Machine-Based Tracking. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics* (New Orleans, Louisiana, USA) *(IWSPA '16)*. Association for Computing Machinery, New York, NY, USA, 33–40. https://doi.org/10.1145/2875475.2875479

[50] Conor Kelton, Matteo Varvello, Andrius Aucinas, and Benjamin Livshits. 2021. Browselite: A Private Data Saving Solution for the Web. *arXiv preprint arXiv:2102.07864* (2021).

[51] Jonathan Lee, Tobias Lidskog, and Peter Hedenskog. 2020. Browsertime. https://www.sitespeed.io/documentation/browsertime/introduction/. Accessed: 2020-02-6.

[52] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. 2016. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th USENIX Security Symposium (USENIX Security 16)*.

[53] Mozilla and individual contributors. 2005-2019. Document Object Model (DOM). https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.

[54] Mozilla and individual contributors. 2005-2019. The HTML DOM API. https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API.

[55] Netmarketshare. 2019. Market Share Statistics for Internet Technologies. https://netmarketshare.com.

[56] 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

[57] Ravi Netravali and James Mickens. 2018. Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 249–266. https://www.usenix.org/conference/nsdi18/presentation/netravali-prophecy

[58] Travis Roman. 2018. JS Blocker. https://jsblocker.toggleable.com/. Accessed: 2020-05-02.

[59] Ashiwan Sivakumar, Vijay Gopalakrishnan, Seungjoon Lee, Sanjay Rao, Subhabrata Sen, and Oliver Spatscheck. 2014. Cloud is not a silver bullet: A case study of cloud-based mobile browsing. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*. 1–6.

[60] Alexander Sjösten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits. 2020. Filter List Generation for Underserved Regions. In *Proceedings of The Web Conference 2020*. 1682–1692.

[61] Peter Snyder, Antoine Vastel, and Benjamin Livshits. 2018. Who filters the filters: Understanding the growth, usefulness and efficiency of crowdsourced ad blocking. *arXiv e-prints* (2018), arXiv–1810.

[62] Matteo Varvello and Benjamin Livshits. 2021. Shedding (Some) Light on Mobile Browsers Energy Consumption. In *TMA*.

[63] 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

[64] 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

[65] Yao Wang, Wan-dong Cai, and Peng-cheng Wei. 2016. A deep learning approach for detecting malicious JavaScript code. *security and communication networks* 9, 11 (2016), 1520–1534.

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

[67] 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* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 2215–2224. https://doi.org/10.1145/3308558.3313575

[68] Chuan Yue and Haining Wang. 2009. Characterizing Insecure Javascript Practices on the Web. In *Proceedings of the 18th International Conference on World Wide Web* (Madrid, Spain) *(WWW '09)*. Association for Computing Machinery, New York, NY, USA, 961–970. https://doi.org/10.1145/1526709.1526838

[69] Chuan Yue and Haining Wang. 2013. A measurement study of insecure javascript practices on the web. *ACM Transactions on the Web (TWEB)* 7, 2 (2013), 7.

[70] Yasir Zaki, Jay Chen, Thomas Pötsch, Talal Ahmad, and Lakshminarayanan Subramanian. 2014. Dissecting Web Latency in Ghana. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (Vancouver, BC, Canada) *(IMC '14)*. Association for Computing Machinery, New York, NY, USA, 241–248. https://doi.org/10.1145/2663716.2663748