Passive network monitoring and troubleshooting from within the browser: a data-driven approach

Naomi Kirimi, Chadi Barakat, Yassine Hadjadj-Aoul

Passive network monitoring and troubleshooting from within the browser: a data-driven approach

Naomi Kirimi
Massachusetts Institute of Technology
Cambridge, USA

Chadi Barakat
Inria, Université Côte d'Azur
France

Yassin Hadjadj-Aoul
Univ Rennes, Inria, CNRS, IRISA
France

***Abstract*—Despite recent advancements in terms of network performance, end users still face slow web browsing situations, which can have a range of causes, such as a congested Wi-Fi, a bad wireless signal, or a loaded network or end host. It is thus crucial to monitor the network and troubleshoot the specific causes of slow web browsing, as this benefits end users, operators, and internet service providers alike. Various tools attempting to actively troubleshoot the network through the injection of probes exist. However, these tools are, on the one hand, expensive to run and, on the other hand, not general enough to be able to identify the specific cause of web browsing slowness. This paper addresses the problem by proposing a new lightweight passive measurement solution capable of transforming the web performance measurements collected from within the browser into indicators of network performance anomalies. We validate our solution by emulating a controlled network environment with manually injected anomalies; then, we leverage the measurement data available within the browser to build a predictive model that uses a random forest classifier to correctly classify the causes of web browsing performance degradation, with an accuracy of over 95%. This implies that one can build on our solution to propose a tool, in the form of a browser extension, that can be used in the wild to monitor the network and shed light on its anomalies by solely relying on a regular user's web activity.**

***Keywords*—Web browser measurements, passive measurements, network anomalies, supervised machine learning, Wi-Fi emulation, performance degradation, Quality of Experience.**

I. Introduction

Today, web browsing is one of the most widespread services of the Internet. Despite the significant progress made recently in terms of speed and connectivity, the network performance, especially at the access level, still significantly affects the user's Quality of Experience (QoE) while browsing the web. The QoE is the degree of annoyance or frustration with a service [1]. Network performance can also affect the revenue of Internet stakeholders, such as online businesses, which rely heavily on web browsing. There are numerous causes of the degradation of web browsing QoE, such as a user's slow device, bad Wi-Fi configuration, interference from neighbouring Wi-Fi networks, saturation of network access links by other devices, congestion of Internet Service Provider's peering links, overload of the content providers' servers due to an increase in users traffic, and possibly network traffic differentiation by the network operator, to cite a few of them. Actually, the network performance degradation not only affects the user's QoE but also makes it difficult for the user to pinpoint the cause of the degradation. This degradation manifests itself to the user in several forms, starting from an unacceptable delay in the rendering of the web page to a slow interaction with the page after being loaded.

Many tools have been proposed in the literature in an unsuccessful attempt to solve the general problem of network troubleshooting in case of poor web browsing QoE. For instance, Speedtest [2] assesses the speed and delay of a user's internet connection towards a server placed close to where the user is located. MobiPerf [3] and its underlying monitoring library Mobilyzer [4] collect several network performance measurements, such as throughput, latency, DNS lookups and HTTP level measurements on mobile platforms. RTR-NetTest [5] informs users about the current internet service quality, such as their ping and their upload and download speeds. Apple's WiFi Scanner enables a user to scan and monitor signals in the air to discover neighbouring wireless networks and to run Speedtests on any of them. For these tools and many others, the user must install and master them, which quickly becomes cumbersome. They also inject traffic into the network to collect active measurements of its performance. This is known to overload the network and consume resources at a moment when the network itself is in a degraded condition. Moreover, none of these tools provides troubleshooting capabilities outside the part of the network it is customized to (the Wi-Fi part, the path between the user terminal and the closest Speedtest server, etc.). Even though these tools and others are very used in practice, we believe the problem of troubleshooting network performance at a reduced cost and in a portable way is still not totally solved.

Past research also tries to use passive measurements to monitor network performance. In a previous work [6], [7], we propose to solve the problem using web performance metrics present in the browser. Among these metrics, one can find the Connect Start, the time to start the connection with the server; the First Paint (FP), the time when the first pixel is rendered onto the browser; and the Page Load Time (PLT), the time when the web page finishes loading. With the help of machine learning and controlled experimentation, we build an inference tool to predict the Round-Trip Time

This work has been supported by the French government, through the UCA-JEDI Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-15-IDEX-01.

The authors would like to acknowledge the experimental setup developed by Mariella Jreidy during her internship at Inria and that provided the basis for this work.

(RTT) and the download speed of a user's network access. Although these metrics are important for gauging network performance, our previous tool does not provide a solution for troubleshooting network issues or identifying their root causes. To get the data needed to build our machine learning-based inference tool, a set of controlled experiments was carried out where the network conditions were artificially varied and measured, serving as ground truth. In parallel, the web was browsed in these controlled conditions, and web performance measurements were collected. Our solution in [6], [7] has the main benefit of being lightweight and data-driven in the sense that it does not require any probing of the network but solely relies on passive measurements freely available within the browser and related to web page rendering.

In this paper, we build upon this work and propose a new solution following the same approach for identifying the root cause of slow web browsing, and not only for the estimation of the network available resources such as the delay and the speed as we did earlier [6], [7]. In addition to network performance, we experiment with further factors that we believe can impact user's QoE. For instance, the user's device also plays a key role in web QoE. The performance of web browsing is sensitive to low-end hardware of the device, such as the clock frequency and the available memory and CPU [8]. Wi-Fi quality also affects the user's QoE, in particular the Wi-Fi Medium Availability and Interference [9]. Wi-Fi Medium Availability is related to how many resources are available to the user after deducing those used by other concurrent applications and devices. Wi-Fi Interference is related to the amount of wireless noise coming from other sources and perturbing the communication between the mobile device and the base station. The web browsing performance also depends on the capacity and load of the web server(s) serving the page. All these factors must be considered when experimenting with web browsing and building intelligence to troubleshoot network performance issues. Overall, we consider seven classes of possible anomalies ranging from end-user device load to server load passing by Wi-Fi and network performance. As stated above, we follow a controlled experimentation approach seconded by supervised machine learning to transform web performance measurements into indicators on the origin of network anomaly, hoping that the former carries a clear and identifiable fingerprint of each class of the anomalies impacting the end-user QoE. Our code is made open source on github [10]. Our main contribution here is to setup the testbed needed to run the controlled experiments, collect the data, and calibrate a machine-learning model to perform the classification. The results we will show later confirm the feasibility of the approach with an excellent classification accuracy exceeding the 95% for the list of considered scenarios.

In the next section, we present our methodology and the way we collect the dataset needed for our work. Section III summarizes and discusses our results. The paper is concluded in Section IV with some perspectives on our future work.

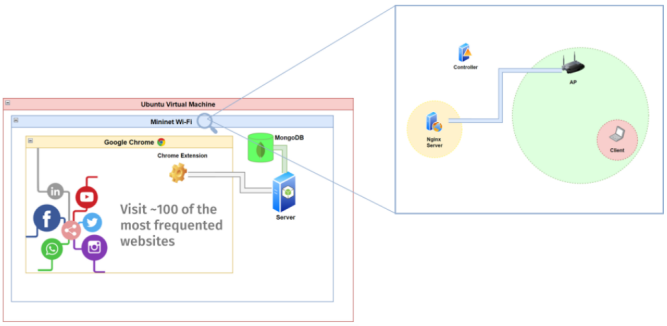

Fig. 1. **System Overview Diagram**. Our testbed models a user connected to a Wi-Fi network and browsing web pages on a server and is connected to the Wi-Fi hotspot via a wired link. We control the bandwidth, delay and packet loss rate of the wired link, as well as the medium availability and interference of the Wi-Fi network.

II. Methodology

Our methodology is based on controlled experimentation where we manually control the introduction of anomalies into the network together with its performance conditions. We emulate a network environment using Mininet Wi-Fi [11], a well-known tool variant of Mininet used in the community to emulate Wi-Fi networks. Mininet Wi-Fi, similarly to Mininet, allows for emulating other network functions, such as bandwidth and delay, as well as the available computing resources on each virtual machine involved in the experiment. Network anomalies are manually introduced into the network one at a time, while several websites are browsed automatically in a serial manner using Chrome browser. Passive measurements available in the browser are collected when the browsing of a page is complete. This collection is done with the help of a Chrome extension plugin, which is launched into the browser during the automated browsing. The collected passive measurements are then paired to network anomaly classes with the help of a supervised machine-learning model that we train to detect and classify the different classes of anomalies we introduced. The obtained model is later used to detect and classify anomalies on the fly, hence providing feedback on the estimated cause of QoE degradation.

Next, we provide a description of our web measurement testbed and the data collected, followed by details on the machine learning part. We first introduce the controlled environment we use to conduct web measurements in Section II-A. We then describe how we introduce anomalies into our controlled environment, run experiments and collect passive web measurements in Section II-B. In Section II-C, we explain how we process the collected data and train a prediction model to detect and classify anomalies.

A. Testbed Overview

1) Mininet Wi-Fi: To collect the dataset needed for the calibration of our classifier, we emulate a communication network with wireless access using Mininet Wi-Fi. Our testbed, described in Fig. 1, models a user connected to a Wi-Fi

network and browsing web pages on a server located behind the Wi-Fi network and connected to the Wi-Fi hotspot via a virtual wired link that we emulate, and that allows us to control the bandwidth, the delay and the packet loss rate of the end-to-end path. Mininet Wi-Fi, a variant of Mininet, functions as a network emulator to create virtual Wi-Fi stations, hosts, controllers, access points, and links [11]. Mininet virtual hosts run standard Linux network software and support research, prototyping, testing, debugging, and other tasks that could benefit from having a complete experimental network on a laptop or a PC. The tool enables complex, yet inexpensive, network topology testing without the need to wire up a physical network and includes a CLI that is topology-aware and a straightforward and extensible Python API for network creation and experimentation [12].

2) Website Duplication and Deployment: We want our web browsing experimentation to be as realistic as possible in a fully controlled environment. For that, we collect the 100 most frequented websites based on the Alexa's ranking [13]. We duplicate the websites using the *wget* tool and store them on a local virtual machine inside our testbed. Out of the 100 targeted websites, we were able to duplicate 75 of them while keeping a good level of similarity with the original websites. The list of the 75 websites that we were able to duplicate is provided in our open-source code on Github [10].

3) Nginx Server: We deploy a local Nginx web server [14] to replay the duplicated websites over our emulated network. Nginx is an open-source web server able to serve a variety of web content including our static HTML pages [14]. We implement Nginx and run it on the virtual server of our testbed located behind the virtual wired link. Once the Nginx server is running locally, the websites can be deployed and automatically browsed from a Chrome browser launched on another virtual machine connected to the Wi-Fi network. We emulate the server using Nginx so as to be able to control every single component in a basic and simple setup.

B. Network Control and Data Collection

We leverage the wealth of measurement data freely available from within the web browser to train our machine-learning model to classify anomalies. This measurement data that we detail next does not require any active probing of the network for its collection since it is of passive type and is automatically available once a website has been browsed on the browser. Further, it can be easily read by any web plugin. We collect this web browsing data after manually introducing anomalies into our Mininet Wi-Fi controlled environment and visiting our duplicated websites one at a time. Once a website is visited and the browser has fully loaded its static home page through the Nginx server, the web browsing data related to this visit is collected and stored. In parallel to data collection from within the browser, we also store in our database the information on the network anomaly we introduce, thus serving as ground truth that we use later to train our machine learning algorithm for network anomaly classification.

1) Introduction of Anomalies: Before browsing the locally duplicated websites on a Chrome browser, we introduce anomalies in our Mininet Wi-Fi experimental environment. We implement one anomaly at a time, and for each anomalous scenario, we run the automated browsing of our duplicated websites in a serial manner while closing the browser between two consecutive visits to reset the browser and remove any measurement interference from other open pages. Our anomalies are implemented by controlling the following four components of our Mininet Wi-Fi testbed: the Wi-Fi access, the wired network, the server, and the user terminal.

On the Wi-Fi level, we introduce two types of anomalies: Medium Availability and Interference. For Medium Availability, we use the *iPerf* tool to generate parallel traffic between two hosts connected to the same Wi-Fi network, hence affecting the Wi-Fi medium availability for our browsing host. With a constant rate UDP traffic of 10 Mbps, 20 Mbps, 30 Mbps, and 40 Mbps, and a Wi-Fi 802.11g technology, we roughly obtain $70\%, 50\%, 30\%$ and 15% of medium availability for each experiment. For Wi-Fi interference, we use the following fading factors proposed by Mininet Wi-Fi: 0, 0.5, 0.75, 1, 2, and 3. The higher the fading factor, the more the signal attenuation, thus leading to packet losses caused by wireless propagation impairments (i.e., other than congestion losses).

On the Network level, we introduce Bandwidth, Packet Loss, and Packet Delay anomalies. For Bandwidth anomalies, we reduce the bandwidth on the emulated link between the web server and the Wi-Fi access point by setting it to the following respective values: 100kbps, 200kbps, 800kbps, 1Mbps and 2Mbps. We introduce Packet Loss anomalies by increasing the packet loss rate on the same link and setting it respectively to 0.5%, 1%, 2%, 3%, 5%, and 10%. Network Delay anomalies are implemented by setting the one-way delay of the wired link to one of these values: 50ms, 80ms, 100ms, 200ms and 300ms. All these anomalies are set in the downlink direction from the web server to the Wi-Fi hotspot, with a total of 16 configurations to experiment with.

On the Web Server level, we overload the server by introducing another station to our emulated network that is directly connected to the server and that sends to it high-rate requests to download the locally duplicated sites. In parallel, we limit the usage of the server's CPU to the following fractions so as to overload the server effectively with three different levels: 0.75, 0.5, and 0.3. This option offered by Mininet puts a limit on the fraction of the CPU the web server has access to. While the server is overloaded, we visit it with our Chrome browser on our emulated Wi-Fi terminal. We follow the same approach in introducing anomalies for the User terminal and limit the CPU utilisation to the following fractions: 0.75, 0.6, 0.5 and 0.3. This gives us a total of 7 experimentation settings for the part related to the CPU load of both user terminal and server.

In total, these different settings give us 10 configurations for the Wi-Fi part, 16 configurations for the wired network part, and 7 configurations for the CPU load part, for a total of 33 configurations we experiment with. For each of these configurations, we browse, in series, the list of web pages and

collect measurements from the browser after each of them is fully loaded. These measurements and information on the type of anomaly introduced form our dataset.

2) Browsing Automation: The browsing and measurement processes are fully automated using the Selenium library [1], the Chrome WebDriver [2] and the PyAutoGUI library[3]. The automation code is made open source on Github [10]. This code is run from the Command Line Interface of the virtual machine with an argument bearing the pathname of the `.txt` file containing the locations of the static HTML pages of the duplicated websites. This prompts the Chrome WebDriver to launch the Chrome browser, install the Chrome plugin and open in series each static HTML web page of the duplicated websites in the browser automatically using the Selenium library. Once a page is fully loaded, the PyAutoGUI clicks the location of the Chrome plugin icon, which is powered by JavaScript to collect the web measurements.

Our Chrome plugin has four main components: (i) a manifest JSON file that specifies aspects of the plugin's functionality, such as name, content, and browser actions for proper running; (ii) the user interface, which is in the form of a pop-up; (iii) the main script file that retrieves the necessary information and handles the calls to different APIs, including the Chrome API, the Network Information API, and the Performance Navigation Timing API; (iv) the server script to communicate with the MongoDB database.

3) Metrics Collection: When PyAutoGUI clicks on the pop-up of the Chrome plugin, the main script file is run and calls the different APIs, resulting in the generation of web measurement metrics. The server script posts the metrics in the MongoDB database deployed locally on the same host. Once all experiments are run with all the respective configurations (33 in total) belonging to our 7 anomaly classes, the measured web metrics can then be extracted from the local database for use in training and testing our target supervised machine-learning model to detect and classify network anomalies from web browsing measurements.

4) Categories of Metrics Created: Upon each website visit, three categories of metrics are collected from the web browser: Network Metrics, System/Device Metrics and Web Metrics. The Network Metrics comprise Effective Connection Type, Downlink and RTT. The Device Metrics comprise CPU Info, which entails (i) CPU Name, (ii) CPU Architecture and (iii) Number of Processors; and System Memory, which entails (i) Available Memory Capacity and (ii) Total Capacity. The Web Metrics comprise Redirect, DNS, Connect, Request, Response, DOM, DOMParse, DOMScripts, ContentLoaded, DomSubRes, Load, RUMSI, Objectindex, HTTP, PLT and TTFB. The metrics collected are stored in the MongoDB database in the following format, whereby the scenario denotes the type of anomaly present when the experiment is run:

- _id: {type: ObjectId}
- scenario: {type: String}
- networkInfo: {type: Object}
- cpuInfo: {type: Object}
- systemInfo: {type: Object}
- webInfo: {type: Object}

In total, we ended up collecting a dataset of 2416 samples, with each sample comprising the measurement data related to the browsing of one website in one abnormal setting.

C. Machine Learning

Given that the main objective of collecting the passive web measurements is to classify anomalies, the main aim of the machine learning model is thus to solve a multi-class classification problem. The classes, in this case, are the types of anomalies we configured, which include Wi-Fi Medium Availability, Wi-Fi Interference, Network Bandwidth, Packet Loss, Delay, User Machine Overload and Server Overload.

1) RUMSI Thresholds: One of the crucial metrics collected from the web browser is the RUMSI (RUM Speed Index), which serves as an indicator of user's Quality of Experience (QoE). We use this indicator to check if the anomaly we introduce in the network is actually impacting the web experience of the user and resulting in an abnormal behavior. In our machine learning part, we only consider those experiments where the speed index points to a slow web browsing.

Regarding the values of the speed index to consider, it is often stated that a speed index of 3 and below indicates that a user's experience would likely range from fair to excellent, while a speed index greater than 3 indicates that the user has potentially a poor QoE [1]. In our case, and to avoid being dependent on one particular value for the speed index threshold, we experiment with different threshold values to illustrate that our model performs excellently across various thresholds. For each threshold value, we only consider abnormal web visits those resulting in a speed index higher than the threshold and train and test our model solely on them. Differently speaking, we test our solution over different levels of user tolerance to network and web anomalies.

Table I summarises the number of data samples for each speed index threshold, while Fig. 2 shows the spread of the samples over the different anomaly classes. Clearly, the higher the threshold, the smaller the number of samples left for training and testing. A threshold of zero means all our samples are considered to be abnormal by the user. We can also notice how our anomaly classes are all present in a balanced way in our dataset for all speed index threshold values. A sharp drop in the number of samples can also be seen for a threshold above 5. So, to make sure that a sufficient number of samples is left for training and testing our solution, we limit ourselves to RUMSI threshold values in the range of 1 to 5.

2) Prediction Model: It is paramount to use a classification algorithm that can easily adapt its complexity to our data. To that end, we use the well-known Random Forest Classifier. A random forest is a collection of trees constructed in a de-correlated manner and are thus advantageous, as they eliminate the high estimation error in case of small changes in the data,

[1]https://www.selenium.dev/documentation/webdriver/
[2]https://chromedriver.chromium.org/
[3]https://pyautogui.readthedocs.io/

TABLE I
DISTRIBUTION OF DATA SAMPLES ACROSS RUMSI THRESHOLDS.

Threshold	0	1	2	3	4	5	10	20	30	40
Number of Data Samples	2416	2372	2047	1583	1157	894	172	25	14	10

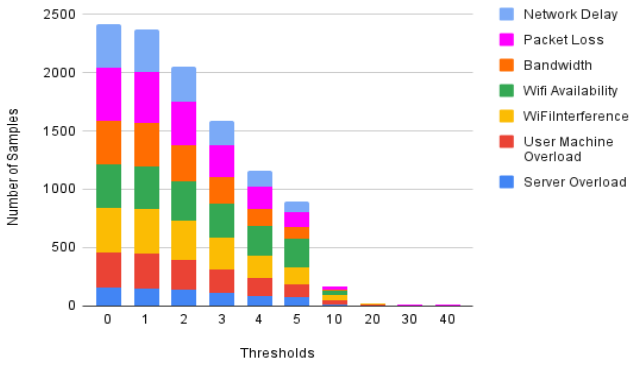

Fig. 2. Distribution of data samples across the anomalies for different values of the RUMSI threshold.

characteristic of a single decision tree. To train our random forest classifier, the data is split into training and test sets, whereby the shares of the data for the sets are 80% and 20%, respectively. The random forest we consider has 100 trees.

III. RESULTS

Next, we present the performance results of our random forest classifier and discuss its capacity to correctly classify the root cause of the anomaly by only using the passive measurements collected from within the browser.

1) Accuracy: Our classifier was found to reach an excellent level of classification accuracy equal to 0.9653, an F1 score of 0.9571 and a recall score of 0.9594 on data samples with RUMSI values greater than 3 seconds. To further illustrate these results, Fig. 3 shows the confusion matrix of our classifier, where rows are the actual anomaly classes and columns are the decisions made by our classifier. We can clearly see how most of the samples lay over the diagonal, pointing to an excellent classification. Only a very few samples are misclassified; their number is less than 4% of the total samples. This confirms our intuition that network anomalies end up leaving their fingerprints in the way the web pages are rendered so that our classifier is able to capture these fingerprints and reflect them afterwards in correct decisions on the type of the anomaly hitting the network.

2) Feature Importance: Table II illustrates the results about feature importance modelled using their information gain, ranked from most important to least important. We dropped those features having identical values across samples due to the low cardinality they present. The dropped features include Web Protocol, Effective Connection Type, System Memory and CPU Information. From the table, we notice how some web-level measurements, such as the time to parse the DOM

TABLE II
FEATURE IMPORTANCE RANKED FROM MOST IMPORTANT TO LEAST IMPORTANT.

Feature
webInfo.domParse (0.190686)
webInfo.response (0.174439)
webInfo.dom (0.121996)
webInfo.domScripts (0.100496)
cpuInfo.cpuProcessors usage load (0.084081)
networkInfo.downlink (0.078662)
webInfo.RUMSpeedIndex (0.077651)
cpuInfo.cpuProcessors user usage (0.050733)
webInfo.pageloadtime (0.029515)
networkInfo.rtt (0.018186)
webInfo.request (0.013993)
webInfo.load (0.013755)
webInfo.dns (0.011148)
webInfo.contentLoaded (0.010479)
cpuInfo.cpuProcessors total usage (0.006402)
webInfo.connect (0.005464)
webInfo.redirect (0.004982)
webInfo.ttfb (0.003806)
webInfo.domSubRes (0.003527)
cpuInfo.cpuProcessors kernel usage (0.000000)

(webInfo.domParse), the time to get the response from the web server (webInfo.response) and the time to get the DOM (webInfo.dom), are holders of precious information that can decide on the class of the network anomaly. Then comes webInfo.domScripts, which is the time the page is ready for user interaction, particularly when the document of the requested web page has finished loading and the document has been parsed but sub-resources such as scripts, images, stylesheets and frames are still loading. Features of type cpuInfo.cpuProcessors account for the number of CPU cycles consumed by the browser and the load of the CPU. Feature NetworkInfo.downlink provides an estimate of the effective bandwidth in megabits per second calculated by the browser based on application layer throughput across recently active connections. In the absence of recent bandwidth measurement data, the attribute value is determined by the properties of the underlying connection technology. Then comes the browser Round-Trip Time estimation, denoted networkInfo.rtt, which is measured by the browser as the time from when the browser sends out a request to when a response is returned. Page Load Time, denoted as webInfo.pageloadtime, is also an important troubleshooting feature modelling the time it takes to download and display the entire content of a web page.

Table III shows the accuracy, F1 and recall scores obtained when our predictive model is trained and tested on data samples with RUMSI thresholds ranging from 1 to 5. Similarly to earlier results for the critical threshold of 3, the model still behaves excellently in classifying the anomalies as we increase

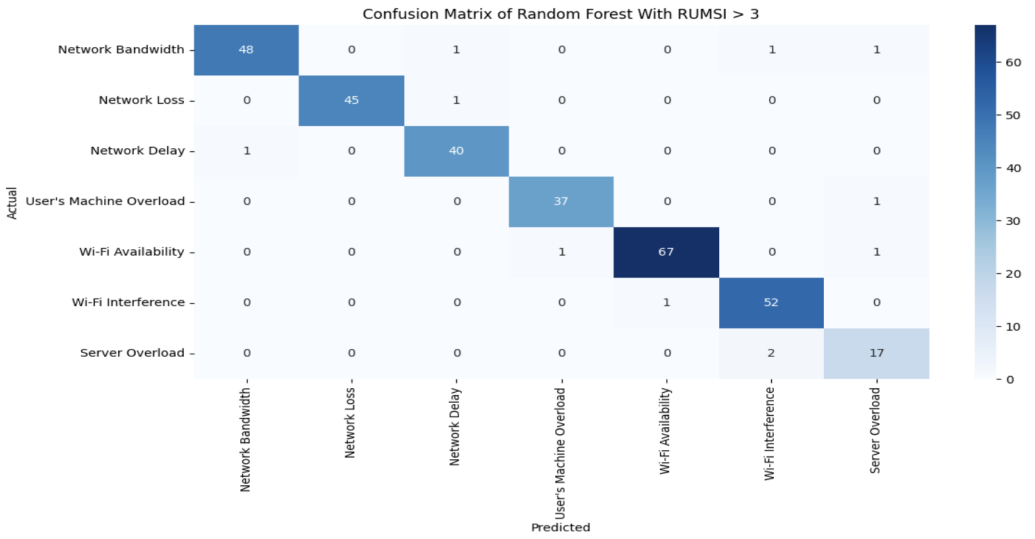

Fig. 3. Confusion Matrix of the anomaly classes for visits with RUMSI value greater than 3. The shaded boxes on the diagonal indicate the number of data samples that were correctly classified. The data samples that were incorrectly classified fall outside the shaded diagonal in each row of the anomaly classes.

TABLE III
PERFORMANCE OF THE RANDOM FOREST CLASSIFIER ACROSS DIFFERENT RUMSI THRESHOLDS.

Threshold	Accuracy	F1 score	Recall score
1	0.9200	0.89632	0.8933
2	0.9415	0.9309	0.9297
3	0.9653	0.9571	0.9594
4	0.9353	0.9168	0.9240
5	0.9218	0.9217	0.9258

the RUMSI threshold. The F1 and recall scores tend even to improve when the threshold gets higher. This is because, for a higher threshold, anomalies are more important, and so are their fingerprints on the browser measurements, thus leading to better classification results.

IV. CONCLUSION

We demonstrate in this paper how passive browser measurements can provide important information regarding network performance. Our classifier can be used in the form of a browser plugin to collect web metrics and troubleshoot the specific cause of network degradation affecting the user's QoE. According to our study, these anomalies could be either User Machine Overload, Server Overload, Wi-Fi Availability, Wi-Fi Interference, Network Bandwidth, Packet Loss or Delay. While the classifier has only been tested in a controlled environment, the excellent results obtained are encouraging enough to test it with further experiments and more advanced anomalies, including, in particular, real wireless access technologies and mobility patterns of the end user. We will also work on deploying the plugin in the wild and test it with real users.

REFERENCES

[1] T. Hoßfeld, F. Metzger, and D. Rossi, "Speed index: Relating the industrial standard for user perceived web performance to web qoe," in *proceedings of the 10th International Conference on Quality of Multimedia Experience (QoMEX)*, pp. 1–6, 2018.

[2] "Speedtest by ookla, the global broadband speed test." https://www.speedtest.net/.

[3] "Mobiperf, an open source application for measuring network performance on mobile platforms." https://www.measurementlab.net/tests/mobiperf.

[4] S. Xu, A. Nikravesh, H. Yao, D. Choffnes, and Z. M. Mao, "Mobilyzer: Mobile network measurement made easy," in *proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, p. 467, 2015.

[5] "Rtr-nettest, a tool to inform users about the current service quality (including upload, download, ping, signal strength) of their internet connection." https://www.netztest.at/en.

[6] I. Taibi, Y. Hadjadj-Aoul, and C. Barakat, "When deep learning meets web measurements to infer network performance," in *proceedings of the 17th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pp. 1–6, 2020.

[7] I. Taibi, Y. Hadjadj-Aoul, and C. Barakat, "Leveraging web browsing performance data for network monitoring: a data-driven approach," in *proceedings of the 2022 IEEE Global Communications Conference (GLOBECOM)*, pp. 6121–6126, 2022.

[8] M. Dasari, C. Kelton, J. Nejati, A. Balasubramanian, and S. R. Das, "Demystifying hardware bottlenecks in mobile web quality of experience," in *proceedings of the SIGCOMM Posters and Demos*, (New York, NY, USA), p. 43–45, 2017.

[9] D. da Hora, K. van Doorselaer, K. van Oost, and R. Teixeira, "Predicting the effect of home wi-fi quality on qoe," in *proceedings of the 2018 IEEE Conference on Computer Communications (INFOCOM)*, 2018.

[10] "Github." https://github.com/chadibarakat/wemon.

[11] R. R. Fontes, S. Afzal, S. H. B. Brito, M. A. S. Santos, and C. E. Rothenberg, "Mininet-wifi: Emulating software-defined wireless networks," in *proceedings of the 2015 11th International Conference on Network and Service Management (CNSM)*, pp. 384–389, 2015.

[12] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, (New York, NY, USA), 2010.

[13] "Alexa Internet." https://www.expireddomains.net/alexa-top-websites/, discontinued on May 1st, 2022.

[14] "Nginx, an open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more." https://nginx.org/.