Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Enhancing Web Applications Observability through Instrumented Automated Browsers[☆]

Boni García [a,*], Filippo Ricca [b], Jose M. del Alamo [c], Maurizio Leotta [b]

[a] *Universidad Carlos III de Madrid, Madrid, Spain*
[b] *Università di Genova, Genova, Italy*
[c] *ETSI Telecomunicación, Universidad Politécnica de Madrid, Madrid, Spain*

ARTICLE INFO

Article history:
Received 18 June 2022
Received in revised form 17 January 2023
Accepted 19 April 2023
Available online 5 May 2023

Keywords:
Browser automation
Log gathering
Empirical study

ABSTRACT

In software engineering, observability is the ability to determine the current state of a software system based on its external outputs or signals such as metrics, logs, or traces. Web engineers rely on the web browser console as the primary tool to monitor the client-side of web applications during end-to-end tests. However, this is a manual and time-consuming task due to the different browsers available. This paper presents BrowserWatcher, an open-source browser extension providing cross-browser capabilities to observe web applications and automatically gather browser console logs in different browsers (e.g., Chrome, Firefox, or Edge). We have leveraged this extension to conduct an empirical study analyzing the browser console of the top-50 public websites manually and automatically. The results show that BrowserWatcher gathers all the well-known log categories such as console or error traces. It also reveals that each web browser additionally includes other types of logs, which differ among browsers, thus providing distinct pieces of information for the same website.

1. Introduction

Observability and monitoring are frequently used interchangeably, although slight differences exist between these terms. On the one hand, IEEE defines monitoring as the process of supervising, recording, analyzing, or verifying the operation of a system or component (IEEE, 1990). On the other hand, observability uses instrumentation tools to provide insights that aid monitoring. In other words, monitoring can happen when a system is observable (Niedermaier et al., 2019).

The term observability comes from the classical control theory by Kalman (1970) and refers to the ability to infer the internal state of a system through the collection and analysis of its external outputs (Niedermaier et al., 2019). It allows understanding the system's internal state by leveraging its external indicators. The three pillars of observability are:

- Metrics: measures of system performance over time, such as response time, transactions per second, or memory usage, to name a few.
- Logs: lines of text (typically timestamped) that a system produces when running a piece of code.
- Traces: representation of causally related distributed events (such as selected logs) that characterize the request flow of a given operation in a software system.

Observability can be essential for maintaining complex software systems and determining the root cause of any issue. For example, in web testing, when an automated end-to-end test fails (e.g., with Selenium WebDriver[1]), and since the whole system is tested, it is usually challenging to discover the underlying error cause (García et al., 2020). A relevant source of information for an end-to-end failed test can be the browser log since it might contain client-side error traces that allow determining the cause of the failure. Nevertheless, log gathering from automated web browsers can be complex due to the lack of proper tooling. Therefore, this process is typically manual, costly, and error-prone.

To mitigate this problem, this paper presents BrowserWatcher, a novel open-source browser extension designed to observe web applications by instrumenting web browsers such as Chrome, Firefox, or Edge. Its features are cross-browser console log gathering, log displaying, tab recording (i.e., capturing the browser viewport as a media file), Content Security Policy (CSP) disabling, and JavaScript/CSS injection. BrowserWatcher can be used as an extension in any browser implementing the WebExtensions Application Programming Interface (API) (Mozilla MDN, 2022a;

[☆] Editor: Earl Barr.
[*] Corresponding author.

E-mail addresses: boni.garcia@uc3m.es (B. García), filippo.ricca@unige.it (F. Ricca), jm.delalamo@upm.es (J.M. del Alamo), maurizio.leotta@unige.it (M. Leotta).

[1] https://www.selenium.dev/documentation/webdriver/

Chrome Team, 2022b) (Chrome, Firefox, Edge, etc.) or integrated with automated testing tools.

To show its applicability and assess its effectiveness we have carried out an experimental study in which the browser logs of the top-50 most popular websites are gathered automatically using the major web browsers. To compare these results with a ground truth, the console of each browser is also collected manually. The comparison reveals that each web browser displays multiple, sometimes different, types of logs (i.e., errors and warnings of different categories) and also shows relevant differences in the logs produced by the same website in different web browsers.

The remainder of this paper is structured as follows. Section 2 explains the background and motivation of this work. Then, Section 3 presents the proposed toolset to enhance observability in automated end-to-end tests in web applications. Section 4 provides experimental proof of the presented approach by analyzing the browser logs of the top-50 most popular websites gathered automatically with our solution and also through manual inspection. In the light of the results obtained in this experiment, Section 5 analyzes the impact of browser log gathering in end-to-end testing and its implications in cross-browser testing. Then, Section 6 presents similar works available in the current literature. Finally, Section 7 summarizes the findings and future steps of this work.

2. Background and motivation

This section presents the Selenium WebDriver architecture, the automated testing library considered in this paper. Then, it provides an example to explain the problem faced that motivates our work.

2.1. Selenium WebDriver

Software testing consists of the dynamic evaluation of a piece of software, called System Under Test (SUT). Automated software testing implies using specific tools and frameworks to implement and execute test cases (or simply tests) against the SUT (Bertolino, 2007). Depending on the size of the SUT, there are different testing levels. Some of the most relevant levels are unit testing (i.e., assessing individual elements, such as classes or methods), integration testing (i.e., testing different units as a combined entity), and end-to-end testing (i.e., testing the whole system through its user interface) (Quadri and Farooq, 2010).

Selenium WebDriver (often known as simply Selenium, like its umbrella project) is an open-source library that allows controlling web browsers programmatically using different languages (such as Java, JavaScript, Python, Ruby, and C#) (García, 2022). In other words, Selenium WebDriver is a library that allows the development of end-to-end tests for web applications (Leotta et al., 2016a). A recent survey about software testing recognized Selenium as the most valuable testing framework, followed by JUnit and Cucumber (Cerioli et al., 2020). All in all, Selenium is frequently considered the de-facto framework for browser automation (García et al., 2020).

Selenium WebDriver uses the native capabilities of each browser to support the automation. For this reason, a test using the Selenium WebDriver API requires an intermediate component called *driver* in the Selenium jargon. Each browser vendor provides a specific driver. For example, the necessary driver to control Chrome with Selenium WebDriver is called chromedriver,[2]

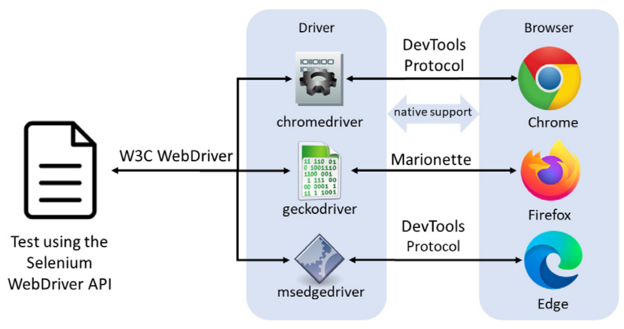


Fig. 1. Selenium WebDriver architecture.

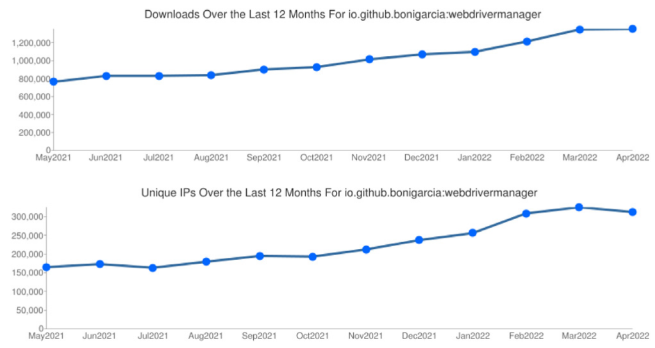


Fig. 2. WebDriverManager usage statistics.

geckodriver[3] for Firefox, or msedgedriver[4] for Edge. A driver is a component that receives incoming messages from Selenium WebDriver tests using a standard protocol called W3C WebDriver (Stewart and Burns, 2022), based on JSON messages over HTTP. The driver translates each W3C WebDriver message to a native command that the browser can understand, such as the DevTools protocol in Chromium-based browsers (such as Chrome and Edge) or Marionette in Firefox. Therefore, as illustrated in Fig. 1, the Selenium WebDriver architecture has three layers composed by the test using the Selenium WebDriver API, the driver, and the browser.

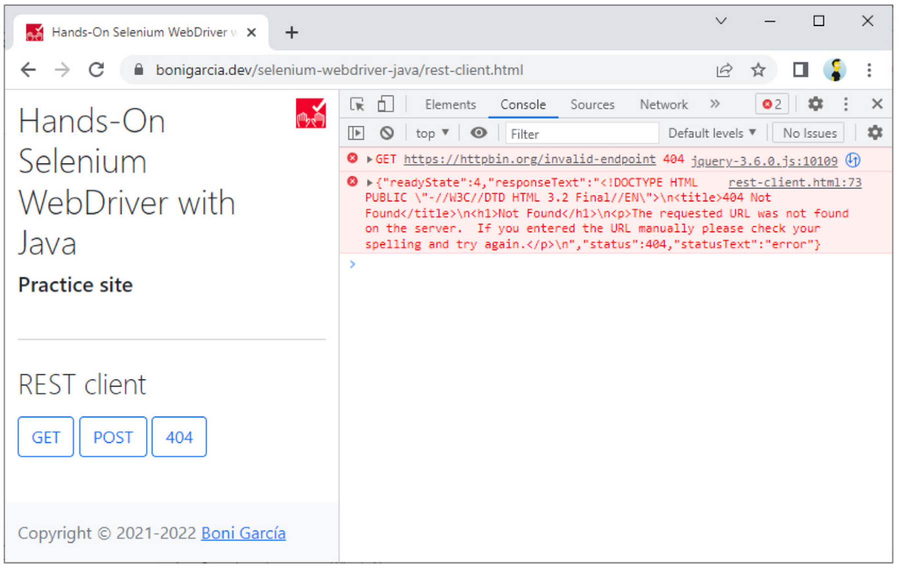
WebDriverManager[5] is an open-source Java helper library for Selenium WebDriver. Its primary feature is automated driver management for the drivers (e.g., chromedriver or geckodriver) required by Selenium WebDriver (García et al., 2021; Leotta et al., 2022). It also discovers browsers installed in the local system, builds `WebDriver` objects (such as `ChromeDriver`, `FirefoxDriver`, etc.), or runs browsers in Docker containers seamlessly. WebDriverManager was first released in 2015 and since then, it has become a well-known helper utility for Selenium WebDriver developers. Fig. 2 shows the evolution of the WebDriverManager monthly downloads and unique IPs from May 2021 to April 2022 according to the Maven Central Sonatype Statistics.[6] For example, WebDriverManager was downloaded 1,354,172 times from 312,621 unique IPs in April 2022.

2.2. Motivational example

As introduced before, logs are one of the pillars of observability. In the end-to-end testing arena, the browser console

2 https://chromedriver.chromium.org/

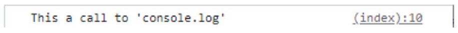
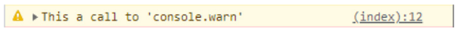
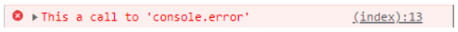
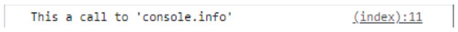
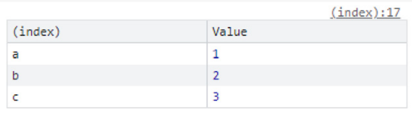
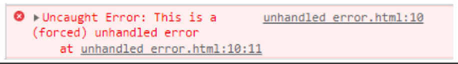
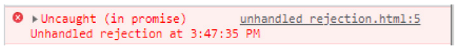
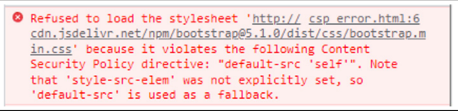
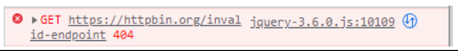
3 https://github.com/mozilla/geckodriver
4 https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/
5 https://bonigarcia.dev/webdrivermanager/
6 https://oss.sonatype.org/


Hands-On Selenium WebDriver
bonigarcia.dev/selenium-webdriver-java/rest-client.html
Elements
Console
Sources
Network
2
top
Filter
Default levels
No Issues
Hands-On Selenium WebDriver with Java
Practice site
REST client
GET
POST
404
Copyright © 2021-2022 Boni García

```
▸GET https://httpbin.org/invalid-endpoint 404 jquery-3.6.0.js:10109
▸{"readyState":4,"responseText":"<!DOCTYPE HTML    rest-client.html:73
PUBLIC \"-//W3C//DTD HTML 3.2 Final//EN\">\n<title>404 Not
Found</title>\n<h1>Not Found</h1>\n<p>The requested URL was not found
on the server.  If you entered the URL manually please check your
spelling and try again.</p>\n","status":404,"statusText":"error"}
```

Fig. 3. Example of log error trace in the browser console.

Table 1
Traceable log categories (i.e., types of logs that BrowserWatcher is able to monitor and gather).

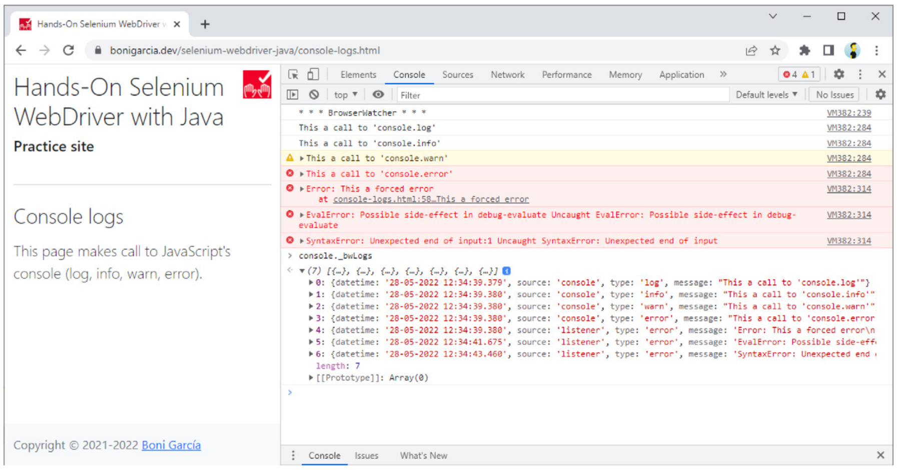
Category	Description	Example
console-log	Regular messages (i.e., calls to `console.log`)	 `This a call to 'console.log'` `(index):10`
console-warn	Warning messages (i.e., calls to `console.warn`)	 `▸This a call to 'console.warn'` `(index):12`
console-error	Error messages (i.e., calls to `console.error`)	 `▸This a call to 'console.error'` `(index):13`
console-info	Informative messages (i.e., calls to `console.info`)	 `This a call to 'console.info'` `(index):11`
console-other	Other console methods, such as `console.dir`, `console.time`, `console.timeEnd`, `console.table`, or `console.count`)	 `(index):17` (index) | Value a | 1 b | 2 c | 3
js-error	JavaScript error traces (e.g., uncontrolled exceptions)	 `▸Uncaught Error: This is a (forced) unhandled error at unhandled_error.html:10:11` `unhandled_error.html:10`
unhandled-rejection	Error when a JavaScript *Promise* with no rejection handler is rejected	 `▸Uncaught (in promise) Unhandled rejection at 3:47:35 PM` `unhandled_rejection.html:5`
csp-violation	Error when a Content Security Policy (CSP) constraint is violated	 `Refused to load the stylesheet 'http://cdn.jsdelivr.net/npm/bootstrap@5.1.0/dist/css/bootstrap.min.css' because it violates the following Content Security Policy directive: "default-src 'self'". Note that 'style-src-elem' was not explicitly set, so 'default-src' is used as a fallback.` `csp_error.html:6`
xhr-error	Error responses from `XMLHttpRequest`	 `▸GET https://httpbin.org/invalid-endpoint 404` `jquery-3.6.0.js:10109`

can contain relevant data to debug and discover the underlying cause of a failed test. Consider the following example. A sample webpage[7] playing the role of REST client makes requests to an external REST service (García, 2022). After each request, the response is displayed in the document body. As illustrated in Fig. 3, one of the requests (the one triggered by the 404 button) uses a non-existing endpoint. As a result, no response is displayed on the page, and an error trace appears in the browser console.

A typical end-to-end test automated with Selenium WebDriver would interact with the page buttons to send requests, waiting for the results to be displayed in the body. In case of error, the test typically fails for a timeout or non-existing element exception (e.g., triggered by a locator expression unable to find the element of interest in the web page (Leotta et al., 2021, 2016b; Nass et al., 2023)). Having access to the browser console would be key to understanding the cause, in this case, a bad request (`404 Not Found`) to a remote endpoint.

This scenario shows an example of a particular error type (labeled as *xhr-error* in Table 1) that can be debugged using the browser console. Moreover, as explained in the upcoming sections, other log categories can contain relevant information for failure analysis. Another typical example is the uncaught exceptions due to JavaScript or CSP violations, to name a few. In addition, warnings can be interesting to gather. These messages

7 https://bonigarcia.dev/selenium-webdriver-java/rest-client.html


Hands-On Selenium WebDriver
bonigarcia.dev/selenium-webdriver-java/console-logs.html
Hands-On Selenium WebDriver with Java
Practice site
Console logs
This page makes call to JavaScript's console (log, info, warn, error).
Copyright © 2021-2022 Boni García
Elements
Console
Sources
Network
Performance
Memory
Application
top
Filter
Default levels
No Issues
Console
Issues
What's New

```
* * * BrowserWatcher * * *                                                    VM382:239
This a call to 'console.log'                                                   VM382:284
This a call to 'console.info'                                                  VM382:284
▸This a call to 'console.warn'                                                 VM382:284
▸This a call to 'console.error'                                                VM382:284
▸Error: This a forced error                                                    VM382:314
    at console-logs.html:58…This a forced error
▸EvalError: Possible side-effect in debug-evaluate Uncaught EvalError: Possible side-effect in debug-   VM382:314
evaluate
▸SyntaxError: Unexpected end of input:1 Uncaught SyntaxError: Unexpected end of input                  VM382:314
> console._bwLogs
< ▾(7) [{…}, {…}, {…}, {…}, {…}, {…}, {…}]
   ▸0: {datetime: '28-05-2022 12:34:39.379', source: 'console', type: 'log', message: "This a call to 'console.log'"}
   ▸1: {datetime: '28-05-2022 12:34:39.380', source: 'console', type: 'info', message: "This a call to 'console.info'"
   ▸2: {datetime: '28-05-2022 12:34:39.380', source: 'console', type: 'warn', message: "This a call to 'console.warn'"
   ▸3: {datetime: '28-05-2022 12:34:39.380', source: 'console', type: 'error', message: "This a call to 'console.error
   ▸4: {datetime: '28-05-2022 12:34:39.380', source: 'listener', type: 'error', message: 'Error: This a forced error\n
   ▸5: {datetime: '28-05-2022 12:34:41.675', source: 'listener', type: 'error', message: 'EvalError: Possible side-eff
   ▸6: {datetime: '28-05-2022 12:34:43.460', source: 'listener', type: 'error', message: 'SyntaxError: Unexpected end
    length: 7
   ▸[[Prototype]]: Array(0)
>
```

Fig. 4. Example of console log gathering through BrowserWatcher.

are symptoms that do not provoke an immediate problem in the SUT but are alerting to a situation that can lead to future problems. Some examples of these warnings are deprecations in the JavaScript or CSS code.

Therefore, gathering logs from automated tests with Selenium WebDriver can be paramount for failure analysis. Moreover, solving warnings in advance is a good practice to prevent future problems. Unfortunately, the W3C WebDriver protocol does not provide any standard mechanism for browser console log gathering. Alternatively, there are some browser-specific mechanisms for log gathering, such as the Chrome DevTools Protocol (CDP) log module (Chrome Team, 2023) or the custom implementation for log gathering (through the browser capability called `goog:loggingPrefs`) implemented by the driver for Chrome (i.e., chromedriver) (Solntsev, 2019). Nevertheless, these non-standard features are unavailable in non-Chromium-based browsers like Firefox. The evolution of W3C WebDriver is called WebDriver BiDirectional (BiDi) protocol (Burns and Smith, 2022), which will support functionality and events related to logging. Unfortunately, BiDi is still in draft status, and its adoption is scarce in browsers and drivers. As a result, automated log gathering is an open issue in improving the observability of web applications in major browsers.

3. Proposed toolset

This paper contributes to the browser automation space by proposing BrowserWatcher, an open-source tool that provides different features to observe web applications. Web engineers can use BrowserWatcher as a regular extension installed on a browser or through instrumented browsers controlled with Selenium WebDriver and managed by WebDriverManager.

3.1. BrowserWatcher

BrowserWatcher[8] is an open-source browser extension explicitly designed to enhance observability in web applications. BrowserWatcher has been implemented on top of the WebExtensions API, a set of cross-browser JavaScript APIs to modify and enhance the browser's capabilities (Mozilla MDN, 2022a; Chrome Team, 2022b). These features provided by BrowserWatcher can be used manually, using its Graphical User Interface (GUI), and programmatically through a JavaScript API. These features are the following:

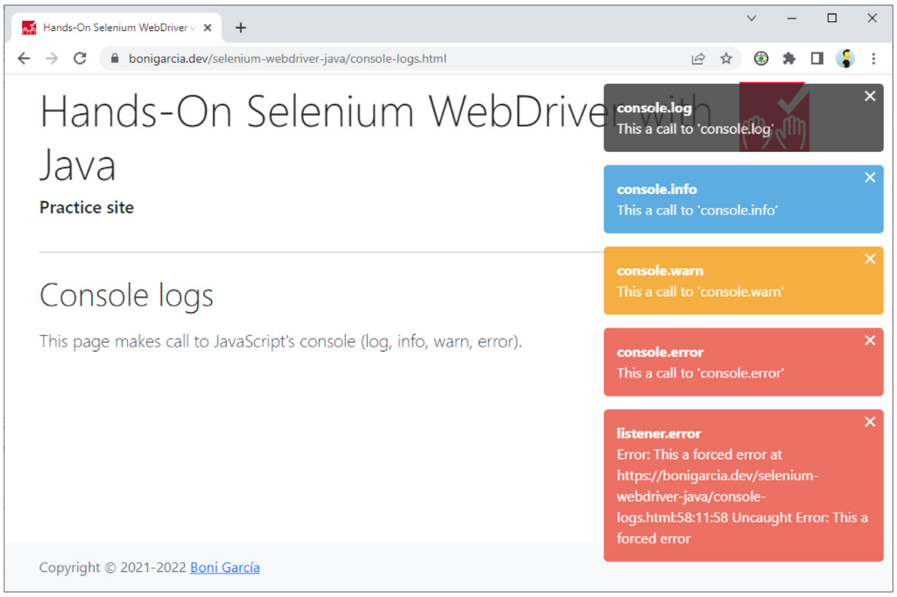
- *Console log gathering*. BrowserWatcher allows gathering different types of browser logs. The data gathered from the console and listeners are stored in a custom property of the `console` object called `_bwLogs`. Fig. 4 shows a screenshot of a sample web page[9] which writes several logs in the console, and how BrowserWatcher gathers this information.
- *Console log displaying*. When this feature is enabled, BrowserWatcher allows displaying the log gathered in the webpage under monitoring as dialog notifications in real-time. Fig. 5 shows a screenshot illustrating it. This feature can be helpful for manual inspection of the generated logs. Also, it can be used in conjunction with tab recording (explained next) to keep track of the logs during automated web navigation.
- *Console tab recording*. BrowserWatcher allows recording what is happening in a browser tab (e.g., when a user is navigating a website) and exporting the recording as a media file using WebM video format. This feature is based on the *tabCapture* API (Chrome Team, 2022c).
- *JavaScript and CSS injection*. BrowserWatcher allows injecting custom JavaScript code, libraries, and CSS stylesheets. This feature allows customizing web pages with external client-side logic and styles (e.g., to track the mouse movements, among many other tuning capabilities).
- *Disabling CSP*. Content Security Policy[10] (CSP) is the name of an HTTP response header aimed at improving the security of a website by instructing the browser about the resources it is allowed to load for that page e.g. allowing images to be loaded from anywhere but restricting a form action to a given endpoint. Nevertheless, developers might want to bypass the CSP headers received from the server for testing purposes (e.g., to inject custom JavaScript code, as explained above, even when CSP is enabled).

Regarding log gathering, BrowserWatcher has been designed to collect well-known log categories, such as console logs or JavaScript error traces. The implementation of this feature is based on JavaScript using the so-called *monkey patching* technique (also known as *runtime hooking*). This technique allows extending or modifying the default behavior of a piece of software at runtime without changing its primary purpose (Hunt, 2019). In particular, BrowserWatcher overrides the JavaScript `console` prototype to monitor the calls of its methods (e.g., `console.log`). Moreover, BrowserWatcher implements different

8 https://bonigarcia.dev/browserwatcher/

9 https://bonigarcia.dev/selenium-webdriver-java/console-logs.html

10 https://content-security-policy.com/


Hands-On Selenium WebDriver with Java

Practice site

Console logs

This page makes call to JavaScript's console (log, info, warn, error).

console.log
This a call to 'console.log'

console.info
This a call to 'console.info'

console.warn
This a call to 'console.warn'

console.error
This a call to 'console.error'

listener.error
Error: This a forced error at https://bonigarcia.dev/selenium-webdriver-java/console-logs.html:58:11:58 Uncaught Error: This a forced error

Copyright © 2021-2022 Boni García

Hands-On Selenium WebDriver w
bonigarcia.dev/selenium-webdriver-java/console-logs.html

Fig. 5. Example of log displaying with BrowserWatcher.

Table 2
WebDriverManager API methods for browser monitoring through BrowserWatcher.

WebDriverManager method	Description
`watch()`	Enable log gathering
`watchAndDisplay()`	Enable log gathering and displaying
`getLogs()`	Read the captured logs
`startRecording()`	Start tab recording
`stopRecording()`	Stop tab recording
`disableCsp()`	Disable CSP headers

JavaScript event listeners that write information in the browser console (e.g., error traces). To implement this feature, the BrowserWatcher extension manifest uses permissions to allow the execution of client-side inline content scripts. These scripts are configured to be executed before any Document Object Model (DOM) element is constructed or any other script is run. This behavior ensures that the patched `console` prototype is always used by all the JavaScript logic handled by the SUT, even for early calls (e.g., for eager page loading strategies, like in Selenium WebDriver (García, 2022)) or when a page change happens (i.e., during manual or automated web navigation). This early loading mechanism can be achieved thanks to the WebExtensions lifecycle (Chrome Team, 2021).

All in all, and given the technical possibilities of JavaScript, BrowserWatcher is capable of gathering a set of log categories. Table 1 summarizes these log categories, called *traceable* logs from now on in this paper.

3.2. WebDriverManager integration

As of version 5.2.0, WebDriverManager is able to monitor web applications through BrowserWatcher. To that aim, WebDriverManager instruments the browsers to be controlled with Selenium WebDriver by installing BrowserWatcher on them at the beginning of the automated session and before the SUT is loaded. Then, WebDriverManager provides several API methods that allow invoking the features provided by BrowserWatcher. Table 2 summarizes these methods. Then, Fig. 6 shows a basic Selenium WebDriver test using the log gathering feature provided by BrowserWatcher through WebDriverManager.

4. Experimentation

We focus on the browser console logs for the experimental validation of this work. Logs are considered one of the pillars of observability. Hence, the information in the browser logs is paramount for developers in failure analysis (also known as troubleshooting) of automated end-to-end tests. Therefore, this empirical study aims to assess the effectiveness of BrowserWatcher in gathering the logs generated by different browsers. From this overall goal, the following Research Questions (RQs) are derived:

RQ1. What are the differences between the logs gathered automatically with BrowserWatcher and WebDriverManager and the actual logs in the browser console?

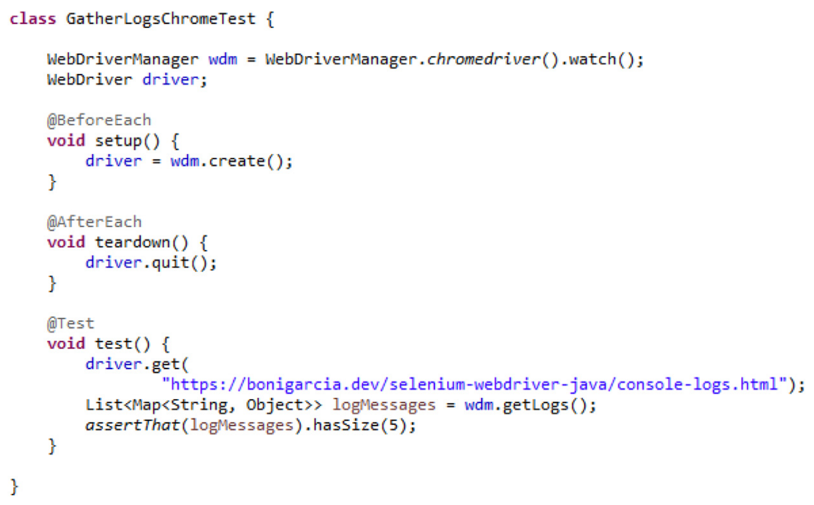
RQ2. Are there any differences between the logs gathered in the major web browsers (both automatically and manually)?

To investigate these RQs quantitatively, we have to implement an end-to-end automated test suite based on Selenium WebDriver and WebDriverManager plus BrowserWatcher to evaluate different SUTs using different browsers. In addition, we need to gather the browser logs manually. We will consider manually collected logs as our ground truth.

4.1. Design and settings

The first aspect we need to decide for designing the experimentation is which browsers are the most used nowadays. We focus on desktop browsers, considering mobile browsers a possible future line. Thus, we can find different online statistics about browser desktop usage. Fig. 7 shows a bar chart of the desktop browser market share worldwide from Apr 2021 to Apr 2022 according to StatCounters.[11] The top-4 desktop browsers, according to these statistics, are Chrome, Safari, Edge, and Firefox. Nevertheless, we cannot use Safari for the automated analysis since the Selenium WebDriver API does not allow installing browser extensions programmatically in Safari (therefore, WebDriverManager does not support it). In conclusion, we focus on the following desktop browsers in the experiment: Chrome, Edge, and Firefox.

11 https://gs.statcounter.com/browser-market-share/desktop/worldwide/#monthly-202104-202204-bar



```java
class GatherLogsChromeTest {

    WebDriverManager wdm = WebDriverManager.chromedriver().watch();
    WebDriver driver;

    @BeforeEach
    void setup() {
        driver = wdm.create();
    }

    @AfterEach
    void teardown() {
        driver.quit();
    }

    @Test
    void test() {
        driver.get(
                "https://bonigarcia.dev/selenium-webdriver-java/console-logs.html");
        List<Map<String, Object>> logMessages = wdm.getLogs();
        assertThat(logMessages).hasSize(5);
    }

}
```

Fig. 6. Example of Selenium WebDriver test using the monitoring capabilities provided by WebDriverManager plus BrowserWatcher.

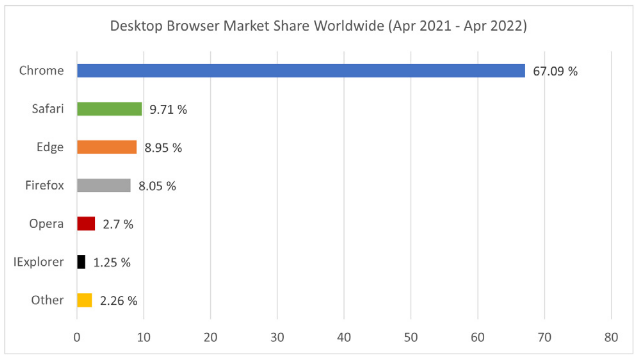


Fig. 7. Desktop browser market share worldwide (Apr 2021–Apr 2022). *Source:* StatCounters.

Another aspect we need to select is the SUT (i.e., the target website) for the experimentation. We aim to gather heterogeneous logs. Thus, we use existing popular websites worldwide as SUT. To that aim, we rely on the information published by Alexa Top Sites, a web service that provides lists of websites ordered by Alexa Traffic Rank (Amazon, 2022). The list of top-50 websites, according to this source on April 19, 2022[12] is presented in Table 3. We use a unique identifier for each website (S01 to S50) to refer to each website in the subsequent sections.

Finally, we implement a Selenium WebDriver automated end-to-end test case using WebDriverManager and BrowserWatcher. This test is parameterized using the cartesian product of the target browsers (Chrome, Edge, and Firefox) and the SUTs (top-50 websites) as arguments. In other words, the test is executed 150 times, one time per browser and website. For each execution, the test performs the following steps:

1. **BrowserWatcher injection**. The test invokes the methods `watch()` and `create()` provided by WebDriverManager. These methods allow the creation of a Selenium WebDriver session in which BrowserWatcher is injected before the DOM is loaded. This way, the browser logs gathering starts from the beginning of the automated session.
2. **Website load**. The test invokes the Selenium WebDriver method `get()` to load the root page of the different SUTs. The default page loading strategy used by Selenium WebDriver (called *normal*) waits until for entire page is loaded. This happens when the DOM *readyState* property is *complete*, meaning that the document and all sub-resources (such as JavaScript files or CSS stylesheets) have finished loading (García, 2022).
3. **Log gathering**. The test invokes the WebDriverManager method `getLogs()` to collect the browser log per execution.
4. **File writing**. The test creates a text file per execution containing the results (e.g., `01_google.com_CHROME.txt`).

The 150 text files generated from the automated tests are the first set of evidence used for data analysis and answering the RQs. In addition to this information, we need to gather the browser console manually for the same input sources. In other words, we need to visit the 50 target websites using Chrome, Edge, and Firefox and collect the console logs manually. As a result, we generated manually another 150 text files.

For replicability, we open-sourced the source code and the generated assets for this experimentation on GitHub.[13] This repository contains the Selenium WebDriver test for the automatic log gathering. In addition, it includes some shell scripts (for Windows and Bash) designed to save time in the manual gathering step by loading the 50 target websites on the different browsers.

The experiment was done on May 3, 2022. The automated and manual log gathering processes were carried out from Madrid (Spain) on a Windows PC with an AMD Ryzen 7 64-bit CPU and 16 GB of RAM. The browsers used for the experiment were Chrome 101.0.4951.67, Edge 101.0.1210.53, and Firefox 100.0.1. The resulting text files containing the gathered logs (both automated and manual) are published in the abovementioned GitHub repository and are analyzed in the following subsection.

4.2. Results

To answer the RQs, we compared every single text file generated in the automated tests (e.g., Chrome browser and S01 website) with its corresponding output of the manual process.

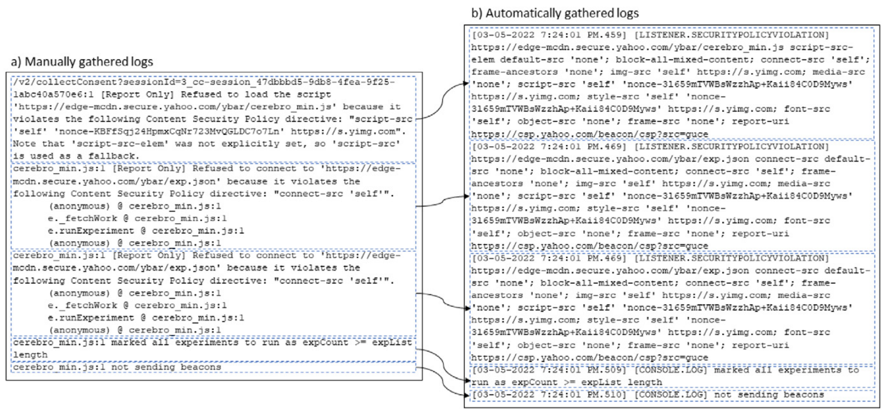
This comparison was a manual process since the logs gathered manually and automatically have different formats, even for the same log entries. Fig. 8 illustrates these differences with

12 https://web.archive.org/web/20220319013139/https://www.alexa.com/topsites

13 https://github.com/bonigarcia/webdrivermanager-log-gathering

Table 3
Top-50 websites according to Alexa Top Sites on April 19, 2022.

Id	Website	Id	Website	Id	Website
S01	google.com	S02	youtube.com	S03	baidu.com
S04	facebook.com	S05	instagram.com	S06	qq.com
S07	bilibili.com	S08	yahoo.com	S09	wikipedia.org
S10	amazon.com	S11	twitter.com	S12	zhihu.com
S13	linkedin.com	S14	reddit.com	S15	whatsapp.com
S16	taobao.com	S17	live.com	S18	chaturbate.com
S19	bing.com	S20	sina.com.cn	S21	sohu.com
S22	csdn.net	S23	tmall.com	S24	weibo.com
S25	zoom.us	S26	microsoft.com	S27	jd.com
S28	github.com	S29	google.com.hk	S30	netflix.com
S31	163.com	S32	office.com	S33	vk.com
S34	canva.com	S35	microsoftonline.com	S36	1688.com
S37	aparat.com	S38	naver.com	S39	twitch.tv
S40	pornhub.com	S41	stackoverflow.com	S42	amazon.in
S43	yahoo.co.jp	S44	xvideos.com	S45	myshopify.com
S46	paypal.com	S47	tiktok.com	S48	aliexpress.com
S49	douban.com	S50	apple.com		



```
/v2/collectConsent?sessionId=3_cc-session_47dbbbd5-9db8-4fea-9f25-
1abc40a570e6:1 [Report Only] Refused to load the script
'https://edge-mcdn.secure.yahoo.com/ybar/cerebro_min.js' because it
violates the following Content Security Policy directive: "script-src
'self' 'nonce-KBFfSqj24HpmxCqNr723MvQGLDC7o7Ln' https://s.yimg.com".
Note that 'script-src-elem' was not explicitly set, so 'script-src'
is used as a fallback.
cerebro_min.js:1 [Report Only] Refused to connect to 'https://edge-
mcdn.secure.yahoo.com/ybar/exp.json' because it violates the
following Content Security Policy directive: "connect-src 'self'".
      (anonymous) @ cerebro_min.js:1
      e._fetchWork @ cerebro_min.js:1
      e.runExperiment @ cerebro_min.js:1
      (anonymous) @ cerebro_min.js:1
cerebro_min.js:1 [Report Only] Refused to connect to 'https://edge-
mcdn.secure.yahoo.com/ybar/exp.json' because it violates the
following Content Security Policy directive: "connect-src 'self'".
      (anonymous) @ cerebro_min.js:1
      e._fetchWork @ cerebro_min.js:1
      e.runExperiment @ cerebro_min.js:1
      (anonymous) @ cerebro_min.js:1
cerebro_min.js:1 marked all experiments to run as expCount >= expList
length
cerebro_min.js:1 not sending beacons
```

```
[03-05-2022 7:24:01 PM.459] [LISTENER.SECURITYPOLICYVIOLATION]
https://edge-mcdn.secure.yahoo.com/ybar/cerebro_min.js script-src-
elem default-src 'none'; block-all-mixed-content; connect-src 'self';
frame-ancestors 'none'; img-src 'self' https://s.yimg.com; media-src
'none'; script-src 'self' 'nonce-31659mTVWBsWzzhAp+Kaii84C0D9Myws'
https://s.yimg.com; style-src 'self' 'nonce-
31659mTVWBsWzzhAp+Kaii84C0D9Myws' https://s.yimg.com; font-src
'self'; object-src 'none'; frame-src 'none'; report-uri
https://csp.yahoo.com/beacon/csp?src=guce
[03-05-2022 7:24:01 PM.469] [LISTENER.SECURITYPOLICYVIOLATION]
https://edge-mcdn.secure.yahoo.com/ybar/exp.json connect-src default-
src 'none'; block-all-mixed-content; connect-src 'self'; frame-
ancestors 'none'; img-src 'self' https://s.yimg.com; media-src
'none'; script-src 'self' 'nonce-31659mTVWBsWzzhAp+Kaii84C0D9Myws'
https://s.yimg.com; style-src 'self' 'nonce-
31659mTVWBsWzzhAp+Kaii84C0D9Myws' https://s.yimg.com; font-src
'self'; object-src 'none'; frame-src 'none'; report-uri
https://csp.yahoo.com/beacon/csp?src=guce
[03-05-2022 7:24:01 PM.469] [LISTENER.SECURITYPOLICYVIOLATION]
https://edge-mcdn.secure.yahoo.com/ybar/exp.json connect-src default-
src 'none'; block-all-mixed-content; connect-src 'self'; frame-
ancestors 'none'; img-src 'self' https://s.yimg.com; media-src
'none'; script-src 'self' 'nonce-31659mTVWBsWzzhAp+Kaii84C0D9Myws'
https://s.yimg.com; style-src 'self' 'nonce-
31659mTVWBsWzzhAp+Kaii84C0D9Myws' https://s.yimg.com; font-src
'self'; object-src 'none'; frame-src 'none'; report-uri
https://csp.yahoo.com/beacon/csp?src=guce
[03-05-2022 7:24:01 PM.509] [CONSOLE.LOG] marked all experiments to
run as expCount >= expList length
[03-05-2022 7:24:01 PM.510] [CONSOLE.LOG] not sending beacons
```

Fig. 8. Example of log comparison (manual vs. automatic) for https://yahoo.com.

a concrete example: the logs gathered for S08, both manually and automatically. As shown in the picture, the content seems completely different. Nonetheless, analyzing its content, it can be checked that it corresponds to the same log entries: three CSP violations and two `console.log` calls. Since this comparison can be an error-prone task, the results were double-checked by the four authors of this paper before further analysis.

We counted the matching log entries in each file, grouping each entry per category. At the end of this process, we found the first relevant finding of this investigation. In addition to the log categories gathered by BrowserWatcher, presented in Table 1 and called *traceable* in this paper, we found other types of logs. From now on, we refer to these categories using the term *non-traceable* in contrast to the traceable logs introduced in Section 3.1. On the one hand, traceable logs can be collected using JavaScript's listeners and monkey-patched objects. On the other hand, to the best of our knowledge, non-traceable logs cannot be gathered using these techniques, and therefore, they are out of the scope of a browser extension like BrowserWatcher.

Furthermore, we discovered that some non-traceable logs appeared in all the browsers under study (Chrome, Edge, and Firefox), and some of them were browser-specific. This way, we group these non-traceable log categories as follows:

- *General*: Log categories found in Chrome, Edge, and Firefox (Table 4).
- *Chromium-specific*: Log categories found both in Chrome and Edge (Table 5).
- *Chrome-specific*: Log categories found only in Chrome (Table 6).
- *Edge-specific*: Log categories found only in Edge (Table 7).
- *Firefox-specific*: Log categories found only in Firefox (Table 8).

Table 9 shows the numeric results of the log gathered both manually and automatically, providing the details about the non-traceable log categories. We use these results to continue with the data analysis and answer the RQs in the following subsections.

4.2.1. RQ1: Automatically vs. manually gathered logs

The traceable logs category is the only type of logs present in the manual and automated processes. Therefore, we can directly compare the results in each browser (Chrome, Edge, and Firefox) in the automated and the manual processes. The following pictures (Fig. 9 for Chrome, Fig. 10 for Edge, and Fig. 11 for Firefox) show a comparison of the matching log entries in the traceable category, both in the automated and the manual analysis.

These results reveal that BrowserWatcher gathers all traceable logs. Nevertheless, we find relevant differences when considering the logs that cannot be collected automatically with BrowserWatcher (i.e., the non-traceable logs) as well. Therefore, considering the whole number of logs (i.e., traceable and non-traceable, as shown in Table 9), 76 entry logs were captured in Chrome out of 220 (i.e., 36% effectiveness), 72 entry logs were captured in Edge out of 239 (i.e., 30.12%), and 145 out of 993 in Firefox (i.e., 14.60%).

The following pictures display these results using bar charts. First, Fig. 12 shows the non-traceable logs that are present in all the target browsers in this study (i.e., Chrome, Edge, and Firefox). Second, Fig. 13 shows the non-traceable logs in the Chromium-based browsers (i.e., Chrome and Edge). Finally, Fig. 14 shows

Table 4
General log categories (i.e., non-traceable types of logs that have been found on Chrome, Edge, and Firefox).

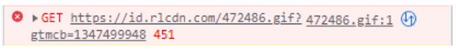
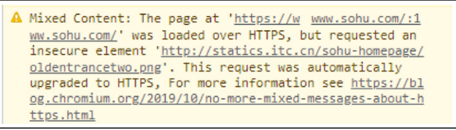
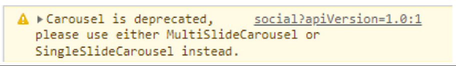
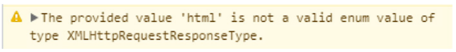
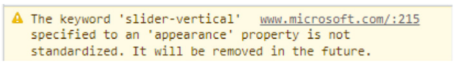
Category	Description	Example
load-error	HTTP errors (e.g., 451 `Unavailable`) when loading resources	 `▸GET https://id.rlcdn.com/472486.gif? 472486.gif:1 gtmcb=1347499948 451`
mixed-content	Load of HTTP resources over an HTTPS session	 `Mixed Content: The page at 'https://w www.sohu.com/:1 ww.sohu.com/' was loaded over HTTPS, but requested an insecure element 'http://statics.itc.cn/sohu-homepage/oldentrancetwo.png'. This request was automatically upgraded to HTTPS, For more information see https://blog.chromium.org/2019/10/no-more-mixed-messages-about-https.html`
js-deprecated	Use of JavaScript deprecated elements	 `▸Carousel is deprecated, social?apiVersion=1.0:1 please use either MultiSlideCarousel or SingleSlideCarousel instead.`
xhr-warn	Warnings when invoking `XMLHttpRequest` `ResponseType` objects	 `▸The provided value 'html' is not a valid enum value of type XMLHttpRequestResponseType.`
css	Warning when applying CSS styles	 `The keyword 'slider-vertical' www.microsoft.com/:215 specified to an 'appearance' property is not standardized. It will be removed in the future.`

Table 5
Chromium log categories (i.e., non-traceable types of logs that have been found on Chrome and Edge).

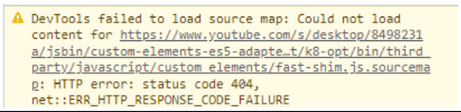
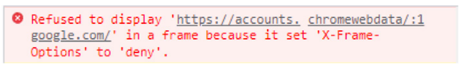
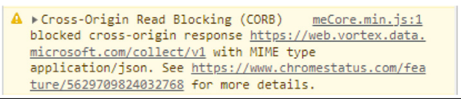
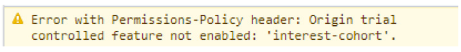
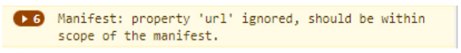
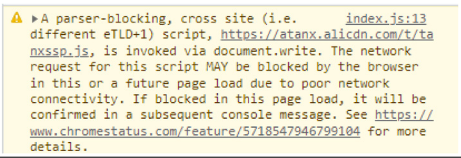
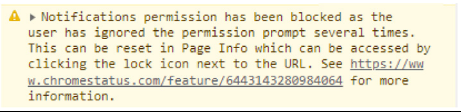
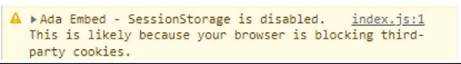
Category	Description	Example
devtools	Problems related to DevTools (Chrome Team, 2022a)	 `DevTools failed to load source map: Could not load content for https://www.youtube.com/s/desktop/8498231a/jsbin/custom-elements-es5-adapte…t/k8-opt/bin/third_party/javascript/custom_elements/fast-shim.js.sourcemap: HTTP error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE`
x-frame	Problems related to the `X-Frame-Options` HTTP response header	 `Refused to display 'https://accounts. chromewebdata/:1 google.com/' in a frame because it set 'X-Frame-Options' to 'deny'.`
corb	Cross-Origin Read Blocking (CORB) blocked responses	 `▸Cross-Origin Read Blocking (CORB) meCore.min.js:1 blocked cross-origin response https://web.vortex.data.microsoft.com/collect/v1 with MIME type application/json. See https://www.chromestatus.com/feature/5629709824032768 for more details.`
header-error	Error with `Permissions-Policy` headers	 `Error with Permissions-Policy header: Origin trial controlled feature not enabled: 'interest-cohort'.`
manifest	Problems with the webapp manifest (Cáceres et al., 2022)	 `▸ 6 Manifest: property 'url' ignored, should be within scope of the manifest.`
cross-origin	Problems with Cross-Origin Resource Sharing (CORS) (Hossain, 2014)	 `▸A parser-blocking, cross site (i.e. index.js:13 different eTLD+1) script, https://atanx.alicdn.com/t/tanxssp.js, is invoked via document.write. The network request for this script MAY be blocked by the browser in this or a future page load due to poor network connectivity. If blocked in this page load, it will be confirmed in a subsequent console message. See https://www.chromestatus.com/feature/5718547946799104 for more details.`

Table 6
Chrome log categories (i.e., non-traceable types of logs that have been found only on Chrome).

Category	Description	Example
notifications-permission	Temporarily stop an origin from requesting a permission	 `▸Notifications permission has been blocked as the user has ignored the permission prompt several times. This can be reset in Page Info which can be accessed by clicking the lock icon next to the URL. See https://www.chromestatus.com/feature/6443143280984064 for more information.`
session-local-storage	Warning due to the session or local storage is disabled	 `▸Ada Embed - SessionStorage is disabled. index.js:1 This is likely because your browser is blocking third-party cookies.`

the non-traceable logs present only in a given browser (Chrome, Edge, or Firefox).

In the light of these results, the answer to RQ1 is two-folded, depending on the log type. On the one hand, the difference between the traceable logs gathered manually and automatically is only syntactic. This fact means that the content of traceable logs automatically gathered is syntactically different from the ground truth (i.e., the manually collected logs), but its content is semantically equivalent (as illustrated, e.g., in Fig. 8). In other words, both sets of logs refer to identical log entries.

Table 7
Edge log categories (i.e., non-traceable types of logs that have been found only on Edge).

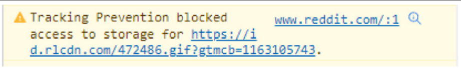
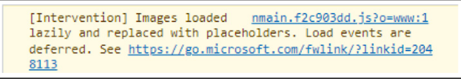
Category	Description	Example
tracking-prevention	Warnings due to tracking prevention, which is Edge's feature that restricts trackers to access browser-based storage and network	 Tracking Prevention blocked access to storage for https://id.rlcdn.com/472486.gif?gtmcb=1163105743. www.reddit.com/:1
intervention	Warnings due to lazy images loading they are likely to be viewed	 [Intervention] Images loaded lazily and replaced with placeholders. Load events are deferred. See https://go.microsoft.com/fwlink/?linkid=2048113 nmain.f2c903dd.js?o=www:1

Table 8
Firefox log categories (i.e., non-traceable types of logs that have been found only on Firefox).

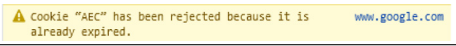
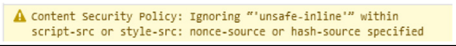
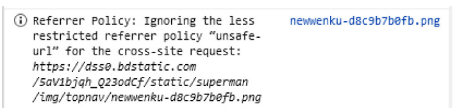
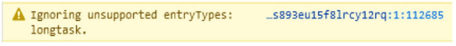
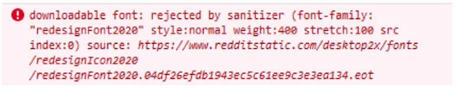
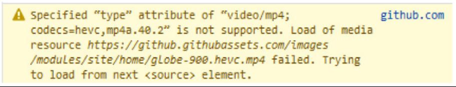
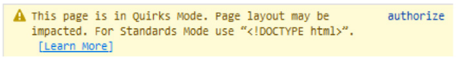
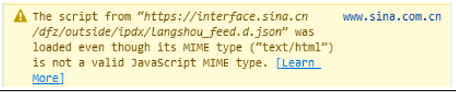
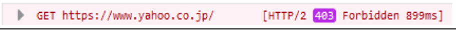
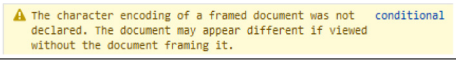
Category	Description	Example
cookies	Warnings due to cookies	 Cookie "AEC" has been rejected because it is already expired. www.google.com
csp-warn	Warnings due to CSP	 Content Security Policy: Ignoring "'unsafe-inline'" within script-src or style-src: nonce-source or hash-source specified
referrer-policy	Use of less restricted policies	 Referrer Policy: Ignoring the less restricted referrer policy "unsafe-url" for the cross-site request: https://dss0.bdstatic.com/5aV1bjqh_Q23odCf/static/superman/img/topnav/newwenku-d8c9b7b0fb.png newwenku-d8c9b7b0fb.png
ignored-unsupported	Problems related to observable performance characteristics (Peña Moreno, 2022)	 Ignoring unsupported entryTypes: longtask. …s893eu15f8lrcy12rq:1:112685
sanitizer	Downloadable font rejected by sanitizer	 downloadable font: rejected by sanitizer (font-family: "redesignFont2020" style:normal weight:400 stretch:100 src index:0) source: https://www.redditstatic.com/desktop2x/fonts/redesignIcon2020/redesignFont2020.04df26efdb1943ec5c61ee9c3e3ea134.eot
media	Unsupported MIME types	 Specified "type" attribute of "video/mp4; codecs=hevc,mp4a.40.2" is not supported. Load of media resource https://github.githubassets.com/images/modules/site/home/globe-900.hevc.mp4 failed. Trying to load from next <source> element. github.com
quirks-mode	Page in Quirks Mode (i.e., a backward compatibility mechanism used in old browsers)	 This page is in Quirks Mode. Page layout may be impacted. For Standards Mode use "<!DOCTYPE html>". [Learn More] authorize
x-content-type	Problems related to the `X-Content-Type- Options` HTTP response header	 The script from "https://interface.sina.cn/dfz/outside/ipdx/langshou_feed.d.json" was loaded even though its MIME type ("text/html") is not a valid JavaScript MIME type. [Learn More] www.sina.com.cn
http2	Problems related to HTTP/2	 GET https://www.yahoo.co.jp/ [HTTP/2 403 Forbidden 899ms]
charset	Problems related to the document character encoding	 The character encoding of a framed document was not declared. The document may appear different if viewed without the document framing it. conditional

Table 9
Number of manual and automated gathered logs in Chrome, Edge, and Firefox.

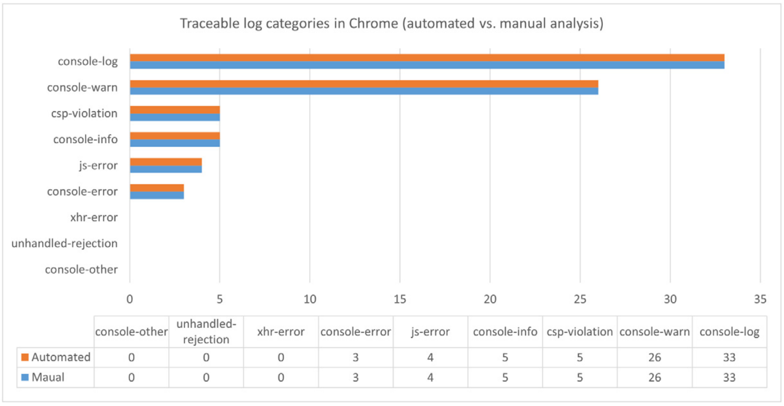
	Automated	Manual						
	Traceable	Traceable	General	Chromium-specific	Chrome-specific	Edge-specific	Firefox-specific	Total
Chrome	76	76	35	103	6	N/A	N/A	220
Edge	72	72	30	59	N/A	76	N/A	239
Firefox	145	145	68	N/A	N/A	N/A	780	993

On the other hand, there are relevant differences when considering all the possible log categories (i.e., the non-traceable logs). Overall, we conclude that the toolset composed of WebDriverManager and BrowserWatcher behaves correctly for the managed (i.e., traceable) log categories. Nevertheless, it shows limitations on other log categories (called non-traceable in this work) as these cannot be collected through JavaScript.

4.2.2. RQ2: Differences in the logs of different browsers

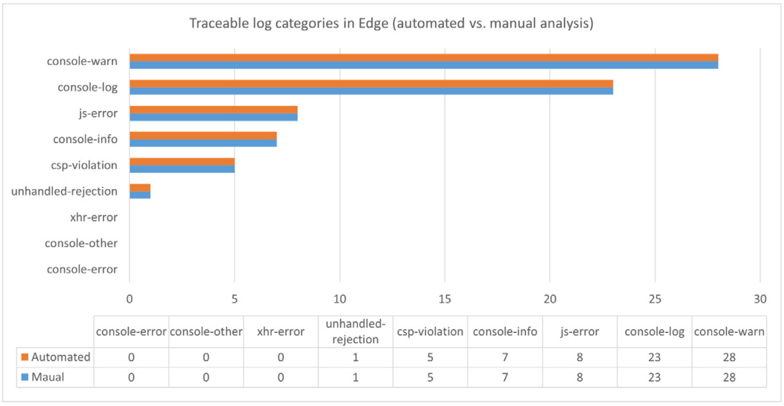
The second RQ investigates the differences between the logs generated in different browsers. We divide the analysis into two parts to answer this question. First, we focus on traceable logs and then we analyze non-traceable ones.

We find relevant differences between the traceable logs generated on different browsers. Fig. 15 illustrates these differences, showing that the number of log entries is different for all the categories. Interestingly, there is a significant distinction: the number of CSP violations. To understand this discrepancy, we check the individual log captures. For instance, Fig. 16 shows the logs available when loading S25 in Chrome, Edge, and Firefox. The browser console in Chrome contains an uncontrolled JavaScript exception and warnings related to the local and session storage. Then, Edge only reports the same JavaScript exception. Finally, the Firefox console is entirely different. Regarding CSP, it contains several warnings and one violation. Although the CSP implementation in the browsers under study are equivalent (Deveria and Schoors,



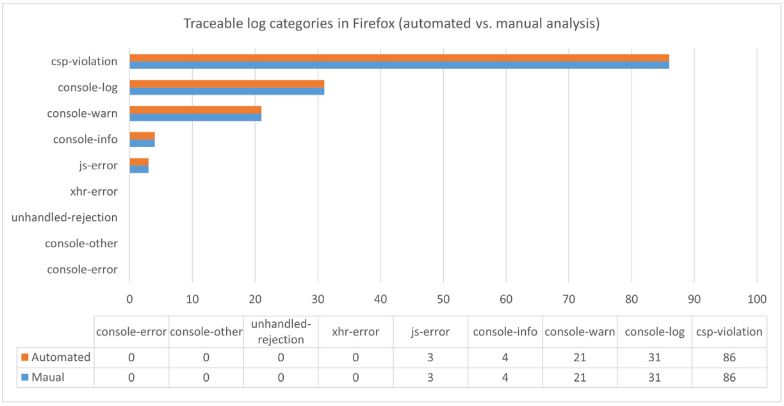
	console-other	unhandled-rejection	xhr-error	console-error	js-error	console-info	csp-violation	console-warn	console-log
🟧 Automated	0	0	0	3	4	5	5	26	33
🟦 Maual	0	0	0	3	4	5	5	26	33

Fig. 9. Traceable log categories in Chrome (automated vs. manual analysis).



	console-error	console-other	xhr-error	unhandled-rejection	csp-violation	console-info	js-error	console-log	console-warn
🟧 Automated	0	0	0	1	5	7	8	23	28
🟦 Maual	0	0	0	1	5	7	8	23	28

Fig. 10. Traceable log categories in Edge (automated vs. manual analysis).



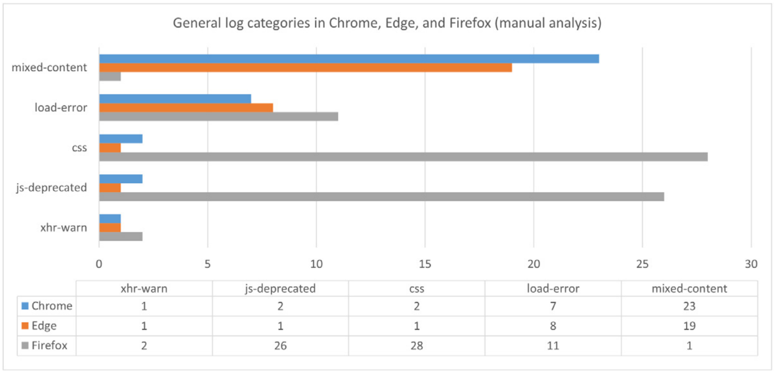
	console-error	console-other	unhandled-rejection	xhr-error	js-error	console-info	console-warn	console-log	csp-violation
🟧 Automated	0	0	0	0	3	4	21	31	86
🟦 Maual	0	0	0	0	3	4	21	31	86

Fig. 11. Traceable log categories in Firefox (automated vs. manual analysis).

2022), Firefox is more verbose (i.e., it displays CSP warnings that Chrome and Edge do not).

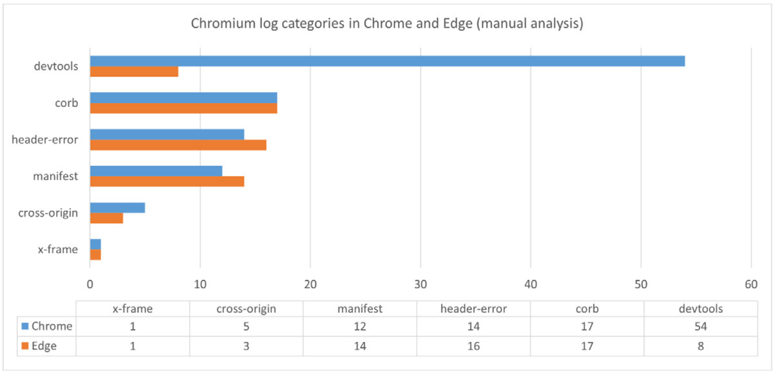
When coming to non-traceable general logs (Fig. 12), there are several relevant differences among browsers. First, Chrome and Edge display many log entries related to *mixed-content* compared to Firefox. On the other side, Firefox displays much more logs related to *css* and *js-deprecation* compared to Chrome and Edge. To illustrate these differences with examples, Fig. 17 shows an example of the *mixed-content* displayed when loading S21. Then Fig. 18 shows the differences related to *css* in S17. Finally, Fig. 19 shows that only Firefox displays problems related to *js-deprecation* when loading S48.

Concerning non-traceable Chromium logs (Fig. 13), the only significant deviation appears in the *devtools* category. This difference is clearly illustrated in Fig. 20, in which Chrome displays more entries related to this category when navigating S12.



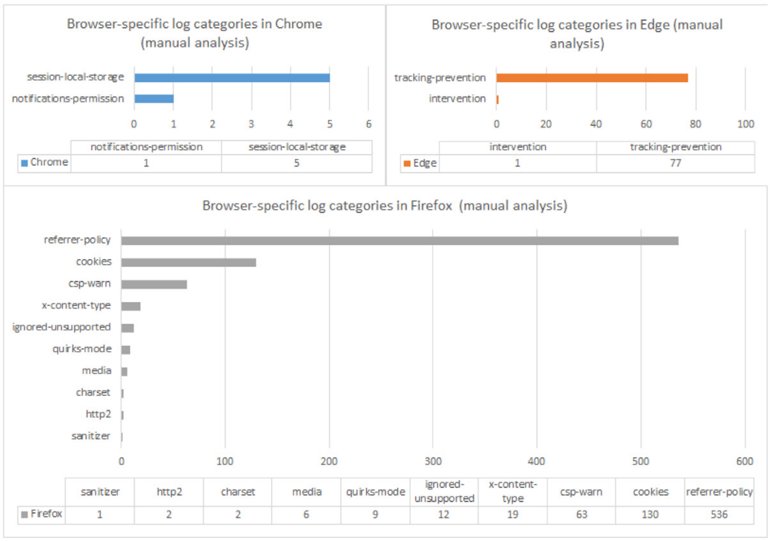
	xhr-warn	js-deprecated	css	load-error	mixed-content
Chrome	1	2	2	7	23
Edge	1	1	1	8	19
Firefox	2	26	28	11	1

Fig. 12. General log categories in Chrome, Edge, and Firefox (manual analysis).



	x-frame	cross-origin	manifest	header-error	corb	devtools
Chrome	1	5	12	14	17	54
Edge	1	3	14	16	17	8

Fig. 13. Chromium log categories in Chrome and Edge (manual analysis).



	notifications-permission	session-local-storage
Chrome	1	5

	intervention	tracking-prevention
Edge	1	77

	sanitizer	http2	charset	media	quirks-mode	ignored-unsupported	x-content-type	csp-warn	cookies	referrer-policy
Firefox	1	2	2	6	9	12	19	63	130	536

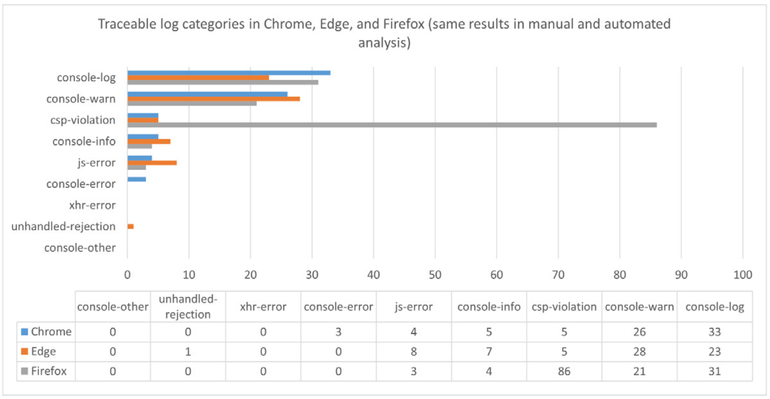
Fig. 14. Browser-specific log categories in Chrome, Edge, and Firefox (manual analysis).

Finally, for non-traceable browser-specific logs (Fig. 14), the most numerous categories appears in Firefox. First, *referrer-policy* (an example can be seen on Fig. 19). Second, *cookies* (for instance, as shown also on Fig. 19). And finally, *csp-warn* (for example, shown in 16).

In conclusion, and answering RQ2, we conclude that there are relevant differences in the logs displayed by the different browsers. First, in the logs collected automatically (i.e., the traceable logs), the main difference occurs in the CSP violations reported by Firefox, which are much more numerous than the CSP errors reported by Chrome and Edge. Second, concerning the logs gathered manually (i.e., the non-traceable logs), there are several log categories with heterogeneous results in the target browsers of this study, such as mixed contents (more numerous in Chrome and Edge than in Firefox), CSS and JavaScript deprecation warnings (reported more times in Firefox than in Chrome and Edge), or DevTools messages (reported more in Chrome than in Edge).

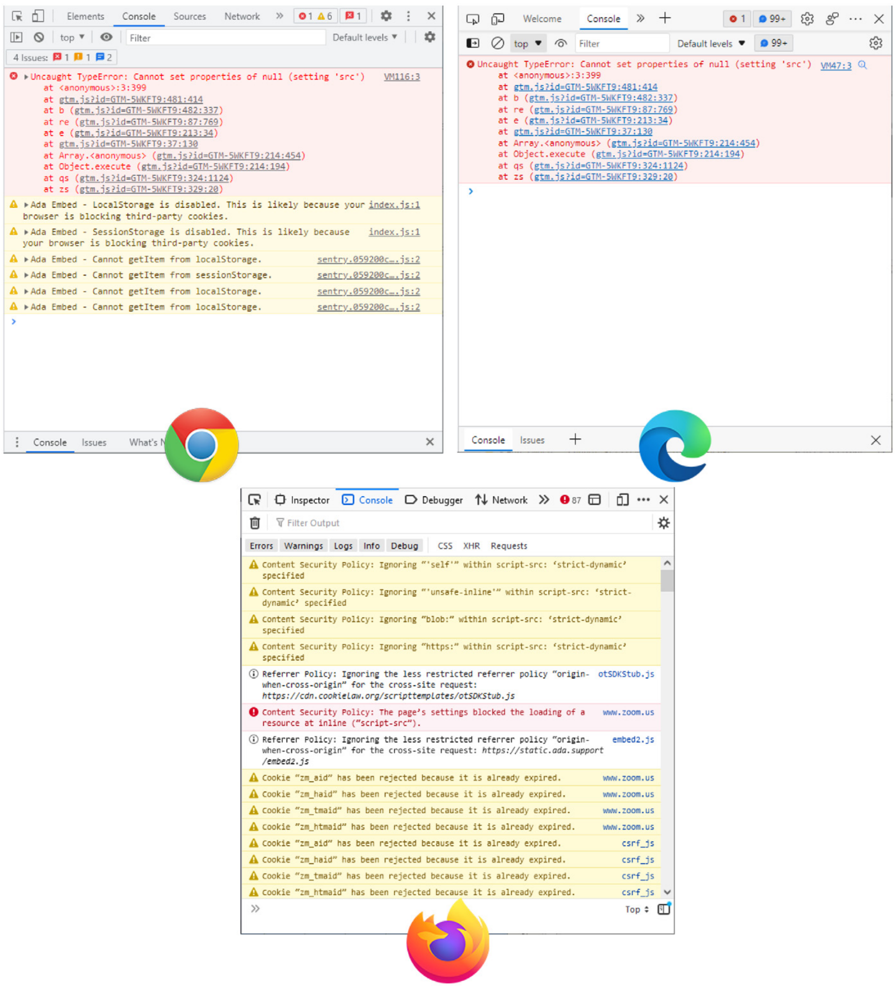
4.3. Threats to the research quality

When assessing the research quality, the reliability and validity of the results need to be considered.



	console-other	unhandled-rejection	xhr-error	console-error	js-error	console-info	csp-violation	console-warn	console-log
■ Chrome	0	0	0	3	4	5	5	26	33
■ Edge	0	1	0	0	8	7	5	28	23
■ Firefox	0	0	0	0	3	4	86	21	31

Fig. 15. Traceable log categories in Chrome, Edge, and Firefox (same results in manual and automated analysis).



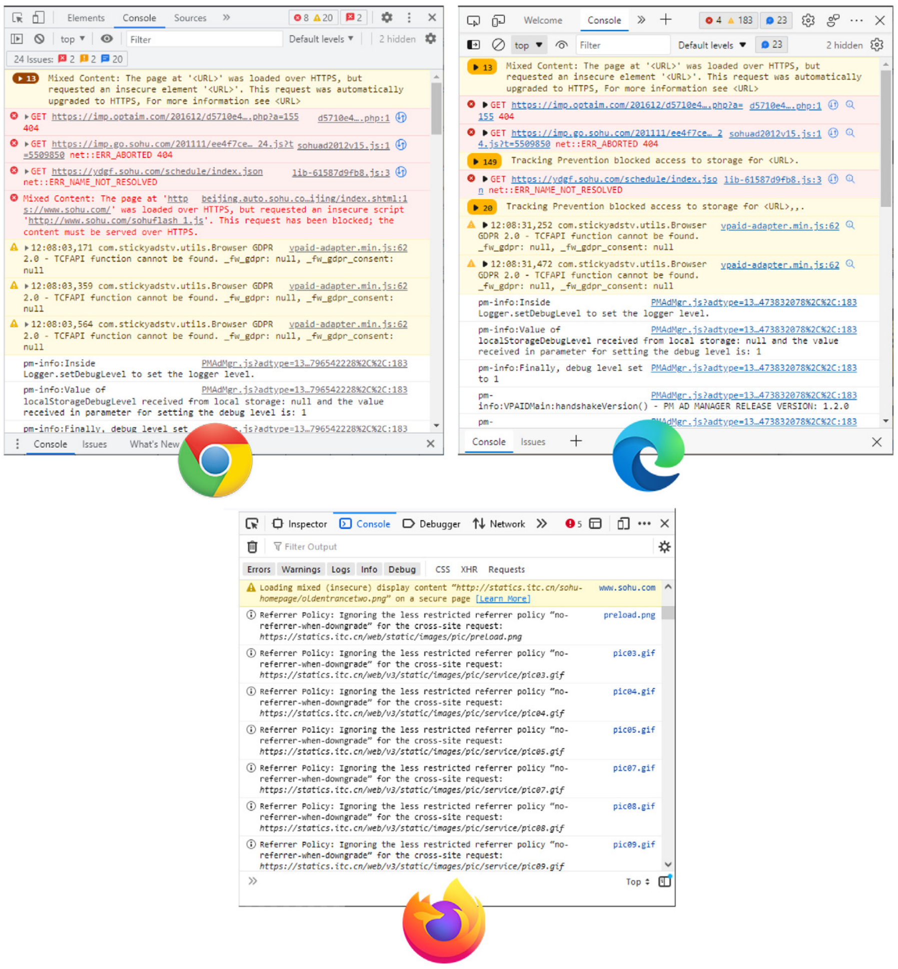
```
▶Uncaught TypeError: Cannot set properties of null (setting 'src')   VM116:3
    at <anonymous>:3:399
    at gtm.js?id=GTM-5WKFT9:481:414
    at b (gtm.js?id=GTM-5WKFT9:482:337)
    at re (gtm.js?id=GTM-5WKFT9:87:769)
    at e (gtm.js?id=GTM-5WKFT9:213:34)
    at gtm.js?id=GTM-5WKFT9:37:130
    at Array.<anonymous> (gtm.js?id=GTM-5WKFT9:214:454)
    at Object.execute (gtm.js?id=GTM-5WKFT9:214:194)
    at qs (gtm.js?id=GTM-5WKFT9:324:1124)
    at zs (gtm.js?id=GTM-5WKFT9:329:20)
▶Ada Embed - LocalStorage is disabled. This is likely because your index.js:1
browser is blocking third-party cookies.
▶Ada Embed - SessionStorage is disabled. This is likely because    index.js:1
your browser is blocking third-party cookies.
▶Ada Embed - Cannot getItem from localStorage.          sentry.059200c….js:2
▶Ada Embed - Cannot getItem from sessionStorage.        sentry.059200c….js:2
▶Ada Embed - Cannot getItem from localStorage.          sentry.059200c….js:2
▶Ada Embed - Cannot getItem from localStorage.          sentry.059200c….js:2
```

```
Uncaught TypeError: Cannot set properties of null (setting 'src')   VM47:3
    at <anonymous>:3:399
    at gtm.js?id=GTM-5WKFT9:481:414
    at b (gtm.js?id=GTM-5WKFT9:482:337)
    at re (gtm.js?id=GTM-5WKFT9:87:769)
    at e (gtm.js?id=GTM-5WKFT9:213:34)
    at gtm.js?id=GTM-5WKFT9:37:130
    at Array.<anonymous> (gtm.js?id=GTM-5WKFT9:214:454)
    at Object.execute (gtm.js?id=GTM-5WKFT9:214:194)
    at qs (gtm.js?id=GTM-5WKFT9:324:1124)
    at zs (gtm.js?id=GTM-5WKFT9:329:20)
```

```
Content Security Policy: Ignoring “'self'” within script-src: ‘strict-dynamic’
specified
Content Security Policy: Ignoring “'unsafe-inline'” within script-src: ‘strict-
dynamic’ specified
Content Security Policy: Ignoring “blob:” within script-src: ‘strict-dynamic’
specified
Content Security Policy: Ignoring “https:” within script-src: ‘strict-dynamic’
specified
Referrer Policy: Ignoring the less restricted referrer policy “origin-   otSDKStub.js
when-cross-origin” for the cross-site request:
https://cdn.cookielaw.org/scripttemplates/otSDKStub.js
Content Security Policy: The page’s settings blocked the loading of a    www.zoom.us
resource at inline (“script-src”).
Referrer Policy: Ignoring the less restricted referrer policy “origin-     embed2.js
when-cross-origin” for the cross-site request: https://static.ada.support
/embed2.js
Cookie “zm_aid” has been rejected because it is already expired.        www.zoom.us
Cookie “zm_haid” has been rejected because it is already expired.       www.zoom.us
Cookie “zm_tmaid” has been rejected because it is already expired.      www.zoom.us
Cookie “zm_htmaid” has been rejected because it is already expired.     www.zoom.us
Cookie “zm_aid” has been rejected because it is already expired.            csrf_js
Cookie “zm_haid” has been rejected because it is already expired.           csrf_js
Cookie “zm_tmaid” has been rejected because it is already expired.          csrf_js
Cookie “zm_htmaid” has been rejected because it is already expired.         csrf_js
```

Fig. 16. Logs in Chrome, Edge, and Firefox when loading https://zoom.us.

To minimize the threat to reliability, our experimental setting is consistent, and our results are reproducible. Firstly, to ensure the consistency of our experiment, we reduced the influence of external factors that might create variation in the results. For example, the same test was run for each website and browser. In the automated process, the test ended once the page was loaded. This is done automatically with Selenium WebDriver by listening to the *complete* Document Object Model (DOM) readiness state (García, 2022). Accordingly, in the manual process the test ended when the browser indicated the page was loaded by checking the spinner displayed in the app icon during the page loading. Admittedly, we cannot ensure the results will be the same across time, as the websites may change. Besides, modern websites can use lazy loading strategies to update web components after loading the DOM. However, to reduce this threat, we carried out manual and automated tests on the same day, with a difference of around one hour between the automated and the manual tests. Also, to minimize the threat to observer consistency, manual and automated tests used *sandboxed* browsers (i.e., using a fresh profile and default configuration per test).


```
Mixed Content: The page at '<URL>' was loaded over HTTPS, but
requested an insecure element '<URL>'. This request was automatically
upgraded to HTTPS, For more information see <URL>
GET https://imp.optaim.com/201612/d5710e4….php?a=155    d5710e4….php:1
404
GET https://imp.go.sohu.com/201111/ee4f7ce… 24.js?t sohuad2012v15.js:1
=5509850 net::ERR_ABORTED 404
GET https://ydgf.sohu.com/schedule/index.json         lib-61587d9fb8.js:3
net::ERR_NAME_NOT_RESOLVED
Mixed Content: The page at 'http   beijing.auto.sohu.co…ijing/index.shtml:1
s://www.sohu.com/' was loaded over HTTPS, but requested an insecure script
'http://www.sohu.com/sohuflash_1.js'. This request has been blocked; the
content must be served over HTTPS.
12:08:03,171 com.stickyadstv.utils.Browser GDPR   vpaid-adapter.min.js:62
2.0 - TCFAPI function cannot be found. _fw_gdpr: null, _fw_gdpr_consent:
null
12:08:03,359 com.stickyadstv.utils.Browser GDPR   vpaid-adapter.min.js:62
2.0 - TCFAPI function cannot be found. _fw_gdpr: null, _fw_gdpr_consent:
null
12:08:03,564 com.stickyadstv.utils.Browser GDPR   vpaid-adapter.min.js:62
2.0 - TCFAPI function cannot be found. _fw_gdpr: null, _fw_gdpr_consent:
null
pm-info:Inside                         PMAdMgr.js?adtype=13…796542228%2C%2C:183
Logger.setDebugLevel to set the logger level.
pm-info:Value of                       PMAdMgr.js?adtype=13…796542228%2C%2C:183
localStorageDebugLevel received from local storage: null and the value
received in parameter for setting the debug level is: 1
pm-info:Finally, debug level set         .js?adtype=13…796542228%2C%2C:183
```

```
Mixed Content: The page at '<URL>' was loaded over HTTPS, but
requested an insecure element '<URL>'. This request was automatically
upgraded to HTTPS, For more information see <URL>
GET https://imp.optaim.com/201612/d5710e4….php?a= d5710e4….php:1
155 404
GET https://imp.go.sohu.com/201111/ee4f7ce… 2 sohuad2012v15.js:1
4.js?t=5509850 net::ERR_ABORTED 404
Tracking Prevention blocked access to storage for <URL>.
GET https://ydgf.sohu.com/schedule/index.jso lib-61587d9fb8.js:3
n net::ERR_NAME_NOT_RESOLVED
Tracking Prevention blocked access to storage for <URL>,,.
12:08:31,252 com.stickyadstv.utils.Browser   vpaid-adapter.min.js:62
GDPR 2.0 - TCFAPI function cannot be found.
_fw_gdpr: null, _fw_gdpr_consent: null
12:08:31,472 com.stickyadstv.utils.Browser   vpaid-adapter.min.js:62
GDPR 2.0 - TCFAPI function cannot be found.
_fw_gdpr: null, _fw_gdpr_consent: null
pm-info:Inside                     PMAdMgr.js?adtype=13…473832078%2C%2C:183
Logger.setDebugLevel to set the logger level.
pm-info:Value of                   PMAdMgr.js?adtype=13…473832078%2C%2C:183
localStorageDebugLevel received from local storage: null and the value
received in parameter for setting the debug level is: 1
pm-info:Finally, debug level set   PMAdMgr.js?adtype=13…473832078%2C%2C:183
to 1
pm-                                PMAdMgr.js?adtype=13…473832078%2C%2C:183
info:VPAIDMain:handshakeVersion() - PM AD MANAGER RELEASE VERSION: 1.2.0
pm-                                   AdMgr.js?adtype=13…473832078%2C%2C:183
```

```
Loading mixed (insecure) display content "http://statics.itc.cn/sohu-   www.sohu.com
homepage/oldentrancetwo.png" on a secure page [Learn More]
Referrer Policy: Ignoring the less restricted referrer policy "no-      preload.png
referrer-when-downgrade" for the cross-site request:
https://statics.itc.cn/web/static/images/pic/preload.png
Referrer Policy: Ignoring the less restricted referrer policy "no-        pic03.gif
referrer-when-downgrade" for the cross-site request:
https://statics.itc.cn/web/v3/static/images/pic/service/pic03.gif
Referrer Policy: Ignoring the less restricted referrer policy "no-        pic04.gif
referrer-when-downgrade" for the cross-site request:
https://statics.itc.cn/web/v3/static/images/pic/service/pic04.gif
Referrer Policy: Ignoring the less restricted referrer policy "no-        pic05.gif
referrer-when-downgrade" for the cross-site request:
https://statics.itc.cn/web/v3/static/images/pic/service/pic05.gif
Referrer Policy: Ignoring the less restricted referrer policy "no-        pic07.gif
referrer-when-downgrade" for the cross-site request:
https://statics.itc.cn/web/v3/static/images/pic/service/pic07.gif
Referrer Policy: Ignoring the less restricted referrer policy "no-        pic08.gif
referrer-when-downgrade" for the cross-site request:
https://statics.itc.cn/web/v3/static/images/pic/service/pic08.gif
Referrer Policy: Ignoring the less restricted referrer policy "no-        pic09.gif
referrer-when-downgrade" for the cross-site request:
https://statics.itc.cn/web/v3/static/images/pic/service/pic09.gif
```

Fig. 17. Logs in Chrome, Edge, and Firefox when loading https://sohu.com.

Secondly, to support the reproducibility of our experiment all the tests, assets, and results are publicly available on GitHub. Furthermore, being integrated within GitHub Actions processes, the experiment is automatically reproduced in a Continuous Integration (CI) build.[14]

We took different measures to address the other threats to validity. Firstly, we conducted an appropriate sampling, working with the top-50 websites worldwide and the major browsers that allow automated testing, representing more than 84% market share. We acknowledge that the results obtained are not generalizable to other browsers though (e.g. Safari), as this requires further work we plan to address once these browsers adopt the upcoming WebDriver BiDi protocol.

Secondly, to address the construct validity, and due to the lack of comparable studies, we manually collected a ground truth to compare our automated results. This fact is in itself a relevant contribution other researchers can leverage upon as this data set is also available on GitHub. Furthermore, different actions were taken to avoid bias when comparing the logs collected manually and automatically. First, the log categories were discussed by all the team members until a common understanding of all the covered aspects was reached. Finally, the classification and comparison of logs was double-checked by the four authors.

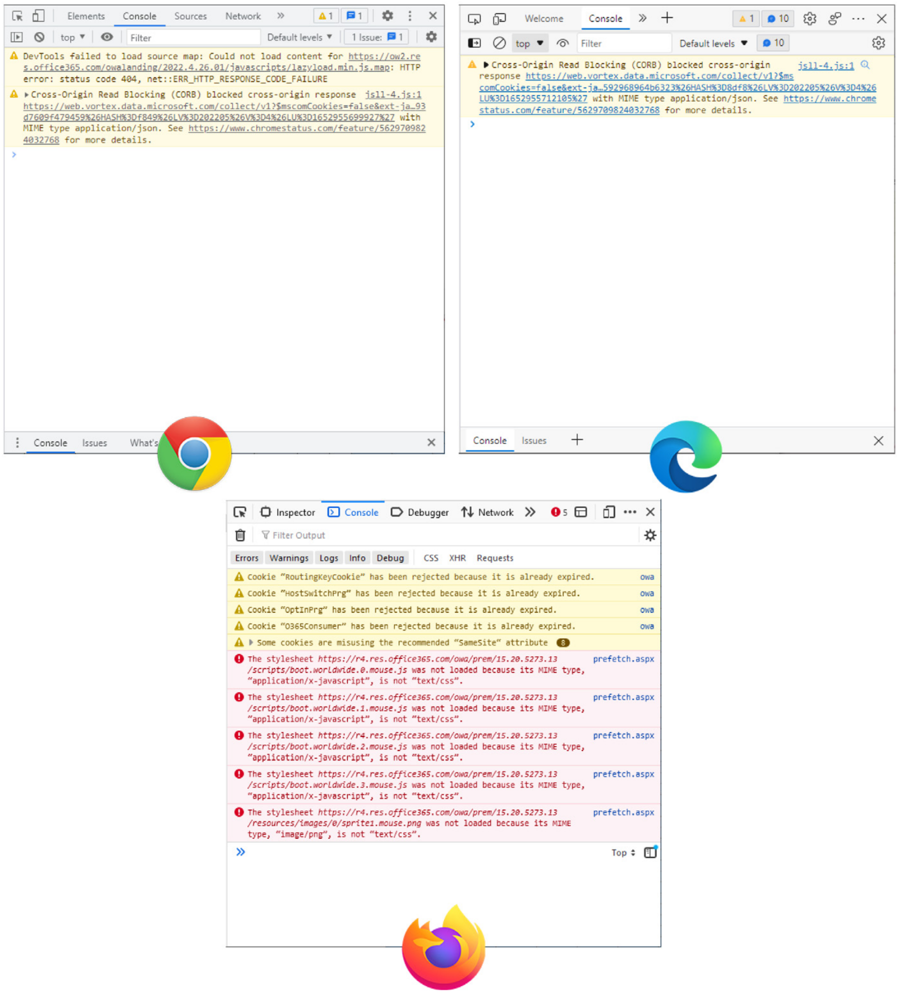
[14] https://github.com/bonigarcia/webdrivermanager-log-gathering/actions

5. Discussion

This work is an effort to improve the observability capabilities of web applications through a cross-browser extension called BrowserWatcher. This extension has been built on top of the WebExtensions API. This way, any modern web browser can benefit from its observability features, such as log gathering or tab recording.

Among all the features provided by BrowserWatcher, and because of its importance in failure analysis, we focused on the capability for gathering browser logs in this paper. Log gathering is not part of the standard mechanism for browser automation (i.e., the W3C WebDriver protocol). Therefore, there is no cross-browser mechanism widely adopted to date for performing this task in automated tests.

To overcome this problem, as of version 5.2.0, WebDriverManager allows instrument web browsers with BrowserWatcher, providing a simple API to gather logs from automated Selenium WebDriver tests. The main benefit of this approach is the seamless implementation of cross-browser log gathering (including in Firefox, which was not possible to date). Thanks to the experimental validation, we checked that the well-known log categories (such as console and error traces) are collected seamlessly with this toolset. Nevertheless, the proposed approach has a significant limitation. Since it is based on JavaScript APIs, there are several log categories (e.g., related to DevTools, mixed content, or CSS,


```
DevTools failed to load source map: Could not load content for https://ow2.re
s.office365.com/owalanding/2022.4.26.01/javascripts/lazyload.min.js.map: HTTP
error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
▸Cross-Origin Read Blocking (CORB) blocked cross-origin response  jsll-4.js:1
https://web.vortex.data.microsoft.com/collect/v1?$mscomCookies=false&ext-ja…93
d7609f479459%26HASH%3Df849%26LV%3D202205%26V%3D4%26LU%3D1652955699927%27 with
MIME type application/json. See https://www.chromestatus.com/feature/562970982
4032768 for more details.
```

```
▸Cross-Origin Read Blocking (CORB) blocked cross-origin     jsll-4.js:1
response https://web.vortex.data.microsoft.com/collect/v1?$ms
comCookies=false&ext-ja…592968964b6323%26HASH%3D8df8%26LV%3D202205%26V%3D4%26
LU%3D1652955712105%27 with MIME type application/json. See https://www.chrome
status.com/feature/5629709824032768 for more details.
```

```
Cookie "RoutingKeyCookie" has been rejected because it is already expired.   owa
Cookie "HostSwitchPrg" has been rejected because it is already expired.      owa
Cookie "OptInPrg" has been rejected because it is already expired.           owa
Cookie "O365Consumer" has been rejected because it is already expired.       owa
▸ Some cookies are misusing the recommended "SameSite" attribute 8
The stylesheet https://r4.res.office365.com/owa/prem/15.20.5273.13       prefetch.aspx
/scripts/boot.worldwide.0.mouse.js was not loaded because its MIME type,
"application/x-javascript", is not "text/css".
The stylesheet https://r4.res.office365.com/owa/prem/15.20.5273.13       prefetch.aspx
/scripts/boot.worldwide.1.mouse.js was not loaded because its MIME type,
"application/x-javascript", is not "text/css".
The stylesheet https://r4.res.office365.com/owa/prem/15.20.5273.13       prefetch.aspx
/scripts/boot.worldwide.2.mouse.js was not loaded because its MIME type,
"application/x-javascript", is not "text/css".
The stylesheet https://r4.res.office365.com/owa/prem/15.20.5273.13       prefetch.aspx
/scripts/boot.worldwide.3.mouse.js was not loaded because its MIME type,
"application/x-javascript", is not "text/css".
The stylesheet https://r4.res.office365.com/owa/prem/15.20.5273.13       prefetch.aspx
/resources/images/0/sprite1.mouse.png was not loaded because its MIME
type, "image/png", is not "text/css".
```

Fig. 18. Logs in Chrome, Edge, and Firefox when loading https://live.com.

among others) that remain uncollected. We believe that these logs, called non-traceable in this paper, should be considered by practitioners (such as researchers and developers) in future work related to client-side browser logs.

We use the presented toolset (i.e., WebDriverManager plus BrowserWatcher) to browse popular websites with different browsers automatically. The differences found in the gathered logs revealed that web applications behave differently depending on the web browser used to render them. Therefore, in our opinion, cross-browser testing (i.e., using different browsers for end-to-end testing) is essential to ensure consistent web application behavior across all browsers.

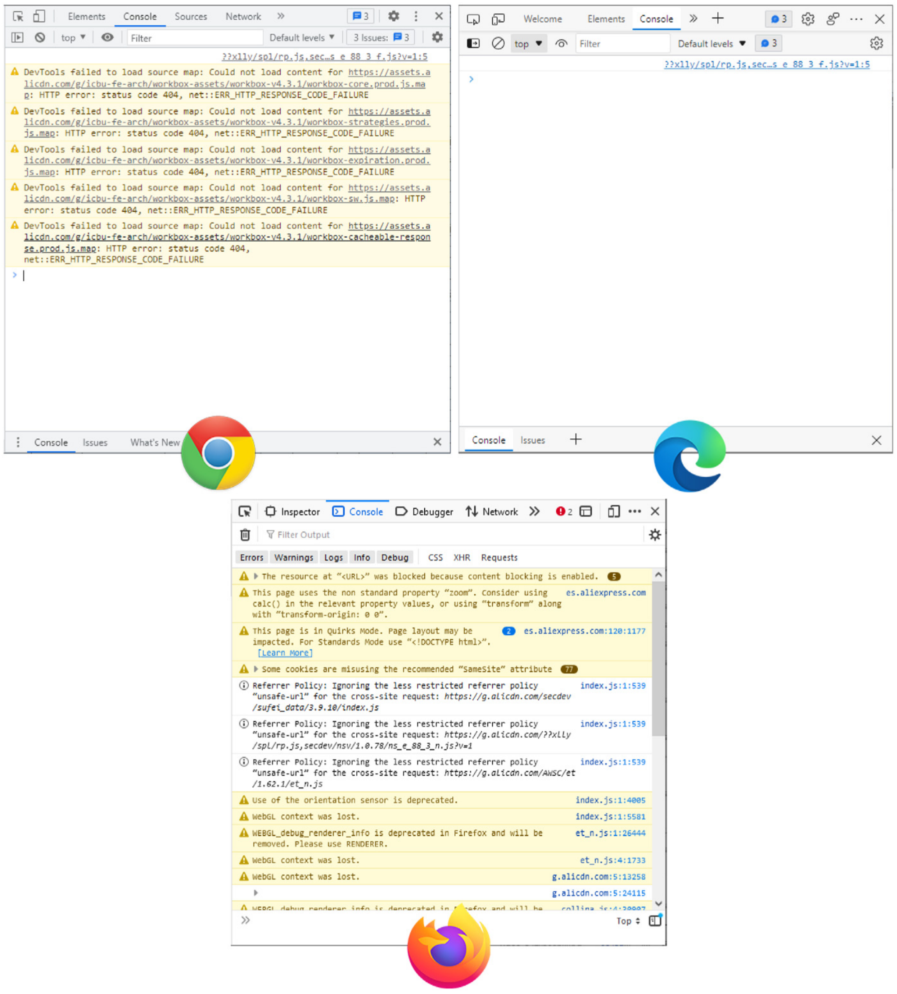
This paper focused on Selenium WebDriver as the browser automation tool. Nevertheless, other browser automation technologies (such as Cypress, Playwright, or Puppeteer) can also use BrowserWatcher. For instance, when using Cypress (which uses local browsers), developers can install BrowserWatcher in the browser to be automated with Cypress. Then, a Cypress script can interact with BrowserWatcher using JavaScript (for example, to collect the browser logs or record the automated session). On the other hand, Playwright and Puppeteer scripts can use BrowserWatcher similarly to Selenium WebDriver, i.e., installing the BrowserWatcher using its own API and interacting with the extension using the BrowserWatcher JavaScript API.

As a final remark, it is worth mentioning that the standard solution for gathering the browser console is currently being developed at the time of this writing. The evolution of the W3C WebDriver protocol is the W3C WebDriver BiDi protocol. BiDi has a module for log gathering, implemented thanks to the bidirectional communication between driver and browser. The first browser supporting BiDi has been Firefox. As of version 101 (released on May 31, 2022), a WebDriver BiDi session can be requested using the classic WebDriver protocol (Mozilla MDN, 2022b). Hence, we plan to support the log gathering in WebDriverManager using the W3C BiDi protocol in future releases. The idea is to enhance the log gathering feature in WebDriverManager to be as complete as possible. To that aim, we plan to make this feature configurable between BrowserWatcher (cross-browser, based on JavaScript), the `goog:loggingPrefs` capability and CDP (only available for Chromium-based browsers), and finally, through BiDi (the standard option in the future).

Nevertheless, it is unclear if BiDi will provide a comprehensive mechanism to gather all types of logs, including the non-traceable categories found in this piece of research. We believe these logs deserve further attention in future research (e.g., in client-side web observability) and development (e.g., to be considered in the BiDi standard).

6. Related works

Although the error logs occurring on browser console are held in high regard by practitioners during testing and debugging sessions, there are few scientific papers dealing with them. By searching the web, it is possible to find various web articles, blogs

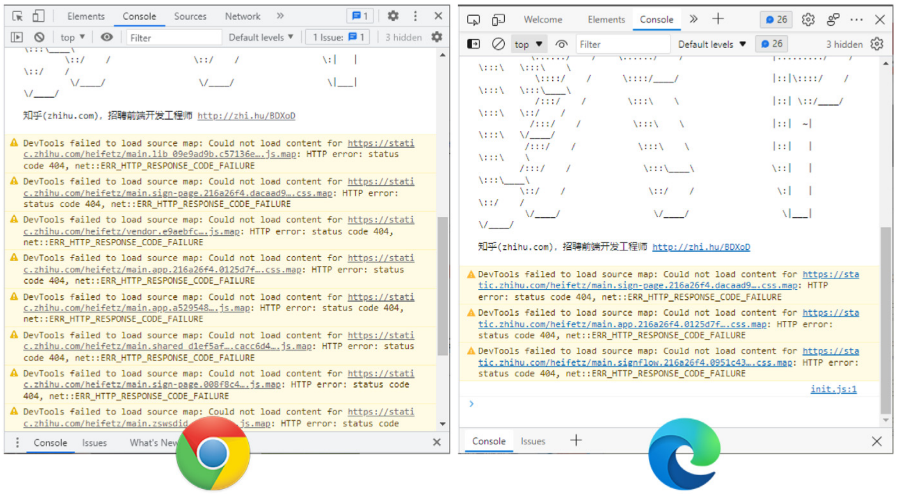


```
??xlly/spl/rp.js,sec…s_e_88_3_f.js?v=1:5
⚠ DevTools failed to load source map: Could not load content for https://assets.a
  licdn.com/g/icbu-fe-arch/workbox-assets/workbox-v4.3.1/workbox-core.prod.js.ma
  p: HTTP error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
⚠ DevTools failed to load source map: Could not load content for https://assets.a
  licdn.com/g/icbu-fe-arch/workbox-assets/workbox-v4.3.1/workbox-strategies.prod.
  js.map: HTTP error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
⚠ DevTools failed to load source map: Could not load content for https://assets.a
  licdn.com/g/icbu-fe-arch/workbox-assets/workbox-v4.3.1/workbox-expiration.prod.
  js.map: HTTP error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
⚠ DevTools failed to load source map: Could not load content for https://assets.a
  licdn.com/g/icbu-fe-arch/workbox-assets/workbox-v4.3.1/workbox-sw.js.map: HTTP
  error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
⚠ DevTools failed to load source map: Could not load content for https://assets.a
  licdn.com/g/icbu-fe-arch/workbox-assets/workbox-v4.3.1/workbox-cacheable-respon
  se.prod.js.map: HTTP error: status code 404,
  net::ERR_HTTP_RESPONSE_CODE_FAILURE
>
```

```
??xlly/spl/rp.js,sec…s_e_88_3_f.js?v=1:5
>
```

```
⚠ ▸ The resource at "<URL>" was blocked because content blocking is enabled.  5
⚠ This page uses the non standard property "zoom". Consider using    es.aliexpress.com
  calc() in the relevant property values, or using "transform" along
  with "transform-origin: 0 0".
⚠ This page is in Quirks Mode. Page layout may be      2  es.aliexpress.com:120:1177
  impacted. For Standards Mode use "<!DOCTYPE html>".
   [Learn More]
⚠ ▸ Some cookies are misusing the recommended "SameSite" attribute  77
ⓘ Referrer Policy: Ignoring the less restricted referrer policy        index.js:1:539
  "unsafe-url" for the cross-site request: https://g.alicdn.com/secdev
  /sufei_data/3.9.10/index.js
ⓘ Referrer Policy: Ignoring the less restricted referrer policy        index.js:1:539
  "unsafe-url" for the cross-site request: https://g.alicdn.com/??xlly
  /spl/rp.js,secdev/nsv/1.0.78/ns_e_88_3_n.js?v=1
ⓘ Referrer Policy: Ignoring the less restricted referrer policy        index.js:1:539
  "unsafe-url" for the cross-site request: https://g.alicdn.com/AWSC/et
  /1.62.1/et_n.js
⚠ Use of the orientation sensor is deprecated.                        index.js:1:4005
⚠ WebGL context was lost.                                             index.js:1:5581
⚠ WEBGL_debug_renderer_info is deprecated in Firefox and will be      et_n.js:1:26444
  removed. Please use RENDERER.
⚠ WebGL context was lost.                                             et_n.js:4:1733
⚠ WebGL context was lost.                                         g.alicdn.com:5:13258
  ▸                                                               g.alicdn.com:5:24115
```

Fig. 19. Logs in Chrome, Edge, and Firefox when loading https://live.com.



```
        \::/    /               \::/    /              \:|   |
\::/    /
         \/____/                 \/____/                \|___|
\/____/

知乎(zhihu.com), 招聘前端开发工程师 http://zhi.hu/BDXoD

⚠ DevTools failed to load source map: Could not load content for https://stati
  c.zhihu.com/heifetz/main.lib_09e9ad9b.c57136e….js.map: HTTP error: status
  code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
⚠ DevTools failed to load source map: Could not load content for https://stati
  c.zhihu.com/heifetz/main.sign-page.216a26f4.dacaad9….css.map: HTTP error:
  status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
⚠ DevTools failed to load source map: Could not load content for https://stati
  c.zhihu.com/heifetz/vendor.e9aebfc….js.map: HTTP error: status code 404,
  net::ERR_HTTP_RESPONSE_CODE_FAILURE
⚠ DevTools failed to load source map: Could not load content for https://stati
  c.zhihu.com/heifetz/main.app.216a26f4.0125d7f….css.map: HTTP error: status
  code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
⚠ DevTools failed to load source map: Could not load content for https://stati
  c.zhihu.com/heifetz/main.app.a529548….js.map: HTTP error: status code 404,
  net::ERR_HTTP_RESPONSE_CODE_FAILURE
⚠ DevTools failed to load source map: Could not load content for https://stati
  c.zhihu.com/heifetz/main.shared_d1ef5af….cacc6d4….js.map: HTTP error: status
  code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
⚠ DevTools failed to load source map: Could not load content for https://stati
  c.zhihu.com/heifetz/main.sign-page.008f8c4….js.map: HTTP error: status code
  404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
⚠ DevTools failed to load source map: Could not load content for https://stati
  c.zhihu.com/heifetz/main.zswsdid          .js.map: HTTP error: status code
```

```
\:::\   \:::\    \
          \::::/    /      \::::/____/                 |::|\::::/    /
\:::\   \:::\____\
          /:::/    /        \:::\    \                 |::| \::/____/
\:::\   \::/    /
         /:::/    /          \:::\    \                |::|  ~|
\:::\   \/____/
        /:::/    /            \:::\    \               |::|   |
\:::\    \
       /:::/    /              \:::\____\              \::|   |
\:::\____\
       \::/    /                \::/    /               \:|   |
\::/    /
        \/____/                  \/____/                 \|___|
\/____/

知乎(zhihu.com), 招聘前端开发工程师 http://zhi.hu/BDXoD

⚠ DevTools failed to load source map: Could not load content for https://sta
  tic.zhihu.com/heifetz/main.sign-page.216a26f4.dacaad9….css.map: HTTP
  error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
⚠ DevTools failed to load source map: Could not load content for https://sta
  tic.zhihu.com/heifetz/main.app.216a26f4.0125d7f….css.map: HTTP error:
  status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
⚠ DevTools failed to load source map: Could not load content for https://sta
  tic.zhihu.com/heifetz/main.signflow.216a26f4.0951c43….css.map: HTTP error:
  status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
                                                                   init.js:1
>
```

Fig. 20. Logs in Chrome and Edge when loading https://zhihu.com.

and posts (e.g.,[15] [16]) that explain how to use a console and analyze the errors contained therein. There are also some web articles explaining how to retrieve error logs using the `LogEntries` class and `get()` method provided by the Selenium WebDriver API (e.g.,[17]). We have decided to divide the related works into three sub-sections: error analysis in JavaScript code, approaches able to identify errors and anomalies through console log analysis, and studies related to WebDriverManager.

6.1. JavaScript errors

A work that emphasizes the importance of monitoring client-side web applications is that of Filipe and Araujo (2016). To demonstrate the limitations of current monitoring tools, the authors ran an experiment to reveal web page errors in a sample of 3,000 web sites (selected similarly to us using Alexa — top-ranked websites) including network and JavaScript errors. The results the authors obtained from their analysis are impressive: as many as 16% of the top 1,000 sites have errors in their own resources; less popular sites have even more. The errors considered in that paper are: HTTP errors (4xx and 5xx), broken links and errors related to resources: fonts, style sheets, images and JavaScript. It is interesting to note that a large proportion of errors are due to JavaScript and in particular for the most popular websites to external resources. The authors conclude the paper by presenting three possible different approaches to improve client observability (which they call transparency). Among these approaches the browser extension solution is sketched, which we also considered in our proposal, to get the most important metrics from the web page interaction.

The goal of another empirical work presented by Ocariza et al. (2013) is that of understanding the root causes and impact of JavaScript faults in web applications. They discovered that the majority of JavaScript faults are DOM-related, that means that they are caused by interactions of the JavaScript code with the DOM. The authors conclude that most JavaScript faults originate from developer mistakes committed in the JavaScript code itself, and not in server-side or HTML code, and as consequence JavaScript developers need development/testing tools that can help them to find JavaScript errors and reason about the DOM. Our BrowserWatcher tool that enhances client-side observability is one of the authors' desired tools.

Another work with the aim of analyzing the JavaScript errors shown in the console of a browser (specifically Firefox) is that of Ocariza et al. (2011). They used FireBug, an add-on to the Firefox web browser, nowadays no longed maintained,[18] to catch the error messages. Similarly to us the evaluation set consists of 50 websites from the Alexa top 100 most visited websites. The contributions of this work are varied as well as the implications for developers, testers and tool builders. Among the contributions we mention the systematic methodology they developed to execute web applications in multiple testing speeds, and gather their error messages from which we were partly inspired to carry out our experiment.

6.2. Automated anomaly detection through console logs

Console logs are used not only in web applications but also for automatic anomaly detection in complex systems, for instance composed of hundreds of software components running on thousands of computing nodes (Bao et al., 2018). Differently from our proposal which focuses on the problems of effectively collecting logs from different browsers, in the context of complex systems the major problem lies in the analysis of the logs. Indeed, a huge quantity of run-time data is continuously collected and stored in log files. Analyzing them to identify causes and locations of problems in case of system malfunctions and failures is a complex and useful task. There is thus a great demand for automatic anomaly detection approaches based on log analysis (Bao et al., 2018). As a consequence, several researchers proposed different solutions. To try to solve this problem several approaches are possible, including:

- Rule-based techniques learn rules that capture the normal behavior of a system. An instance that is not covered by any such rule is considered as an anomaly (Chandola et al., 2009).
- Time series analysis-based techniques analyze the entire log as a single sequence of repeating messages and try to find anomalies with time series analysis methods (e.g., Yamanishi and Maruyama (2005)).
- Learning-based techniques rely on Machine learning to detect anomalies. Depending on the type of data involved and the machine learning techniques employed, anomaly detection approaches can be further classified into two sub-categories (He et al., 2016): supervised anomaly detection and unsupervised anomaly detection.

All these techniques could be useful also in the web application context: indeed the top-50 websites considered in this study can be considered complex systems, in particular from a server-side perspective. However, our study focuses on the browsers console logs (so only client-side). During the experimentation we discovered that in general the amount of logs gathered when visiting a single web page is not huge and can be reasonably analyzed by hand (as we did during our empirical study). However, BrowserWatcher allows to gather logs during the execution of test suites where each test script potentially visits dozen of web pages: in that usage scenario the techniques presented in the literature could help to quickly analyze the logs and finding anomalies. Thus, BrowserWatcher can be also considered a fundamental enabler to perform such advanced analyses on client-side console logs.

6.3. WebDriverManager

WebDriverManager, the tool that among other things makes it possible to use BrowserWatcher, has been reported the first time in the Selenium ecosystem survey by García et al. (2020). That survey, conducted in 2019 by 72 participants from 24 countries, revealed how practitioners use Selenium WebDriver in Web testing. Focusing only on the driver management part, the study revealed that 39% circa of the respondents declared to manage driver managers manually, while 35% circa of the respondents declared to carry out this process automatically. The remaining users (i.e., 26% circa) claimed not to know how drivers are managed in their test suites. This result shows the spread of WebDriverManager.

The complete methodology to carry out the driver management process (i.e., download, setup, and maintenance) and the WebDriverManager tool are presented in García et al. (2021). In addition to presenting in detail the architecture and API of WebDriverManager that paper evaluated, by means of a survey, also the usability of WebDriverManager. Respondents found WebDriverManger useful and usable. Nevertheless, the benefits of

[15] https://wordpress.org/support/article/using-your-browser-to-diagnose-javascript-errors/
[16] https://yoast.com/help/how-to-find-javascript-errors-with-your-browsers-console/
[17] https://thoughtcoders.com/blog/selenium-code-to-test-browser-console-error-logs/
[18] https://getfirebug.com/

WebDriverManager (development effort reduction and improved maintainability) were not considered in the survey.

The benefits of WebDriverManager were instead considered in a subsequent work (Leotta et al., 2022) where a controlled experiment with 25 MSc students was conducted. Results of the experiment show that the adoption of WebDriverManager, instead of manual driver management, significantly reduces the time required to setup/update a multi-browser test suite (average saving of more than 33% of the time).

WebDriverManager was mentioned, as a useful tool, by many practitioners also in the context of a survey aimed at understanding the challenges (and the possible solutions) of end-to-end testing with Selenium WebDriver (Leotta et al., 2023).

Finally, in another paper again García et al. (2022) describe Selenium-Jupiter, a JUnit 5 extension for Selenium WebDriver. Selenium-Jupiter aims to ease the development of end-to-end test suites using WebDriverManager and facilitating the integration with Docker. Selenium-Jupiter allows advanced features for cross-browser testing, load testing, and troubleshooting. Now since the new version of WebDriverManager contains Browser Watcher, Selenium-Jupiter will also be enriched with features related to improving observability in web applications.

7. Conclusions

End-to-end automated testing is one of the most popular mechanisms to ensure quality and reduce defects in web applications. Nevertheless, failure analysis in these types of tests can be challenging, especially for complex systems. This paper presented BrowserWatcher, a browser extension that can be used to instrument web browsers and observe web applications. BrowserWatcher is built on top of the WebExtensions API. Internally, it uses monkey-patched objects and listeners to collect browser logs (e.g., console and error traces) while a website is exercised.

We believe this tool can be helpful for many practitioners worldwide. Therefore, to ease its usage and reach a broader range of developers, we integrated it into WebDriverManager, a popular helper tool (mainly used for automated driver management) of the Selenium ecosystem. This way, the observability features provided by BrowserWatcher are available through a straightforward API used to control instrumented browsers programmatically with Selenium WebDriver.

We performed experimental validation of the proposed toolset to assess BrowserWatcher in gathering the logs generated by different browsers (Chrome, Edge, and Firefox) on 50 popular websites. We checked that the log categories (named *traceable* in this paper) managed with the above-mentioned JavaScript capabilities are always gathered with BrowserWatcher. Nevertheless, we discovered other log categories (named *non-traceable* in this paper) that cannot be collected with JavaScript to the best of our knowledge. We consider this an open question that should drive the practitioners' attention in future research and development.

Another finding of the experimental study presented in this piece of research is that the same website can produce different logs in different web browsers. This result reinforces the value of cross-browser testing (i.e., using different types of browsers for end-to-end web testing) to ensure that web applications behave as expected in any browser.

We plan to keep maintaining both BrowserWatcher and WebDriverManager to benefit the test automation community. In future work, we plan to support the log gathering mechanism provided by the W3C BiDi protocol in WebDriverManager when this feature is available in all major browsers. Another possible future research is to extend this work to mobile browsers.

CRediT authorship contribution statement

Boni García: Conceptualization, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Filippo Ricca:** Methodology, Validation, Writing – review & editing. **Jose M. del Alamo:** Conceptualization, Validation, Writing – review & editing. **Maurizio Leotta:** Methodology, Validation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

All the source code and data from the experimental validation is public available in open source GitHub repositories (linked in the paper).

Acknowledgments

This work was partially supported in part by the Ministerio de Ciencia e Innovación-Agencia Estatal de Investigación, Spain (10.13039/501100011033) through the H2O Learn project under Grant PID2020-112584RB-C31, in part by the Madrid Regional Government through the e-Madrid-CM Project, Spain under Grant S2018/TCS-4307, and in part supported by the Comunidad de Madrid and Universidad Politécnica de Madrid, Spain through the V-PRICIT Research Programme Apoyo a la realización de Proyectos de I+D para jóvenes investigadores UPM-CAM, under Grant APOYOJOVENES-QINIM8-72-PKGQ0J. Funding for Article Processing Charge (APC): Universidad Carlos III de Madrid (Read & Publish Agreement CRUE-CSIC 2023).

References

Amazon, 2022. Alexa top sites. https://www.alexa.com/topsites. (Online Accessed 19 April 2022).

Bao, L., Li, Q., Lu, P., Lu, J., Ruan, T., Zhang, K., 2018. Execution anomaly detection in large-scale systems through console log analysis. J. Syst. Softw. 143, 172–186. http://dx.doi.org/10.1016/j.jss.2018.05.016.

Bertolino, A., 2007. Software testing research: Achievements, challenges, dreams. In: Future of Software Engineering. FOSE'07, IEEE, pp. 85–103.

Burns, D., Smith, M., 2022. WebDriver bidi. Editor's draft. https://w3c.github.io/webdriver-bidi/. (Online Accessed 27 May 2022).

Cáceres, M., Christiansen, K.R., Giuca, M., Gustafson, A., Murphy, D., Kostiainen, A., 2022. Web application manifest, W3C working draft. https://https://www.w3.org/TR/appmanifest/. (Online Accessed 29 May 2022).

Cerioli, M., Leotta, M., Ricca, F., 2020. What 5 million job advertisements tell us about testing: a preliminary empirical investigation. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing. pp. 1586–1594.

Chandola, V., Banerjee, A., Kumar, V., 2009. Anomaly detection: A survey. ACM Comput. Surv. 41 (3), http://dx.doi.org/10.1145/1541880.1541882.

Chrome Team, 2021. Content scripts. https://developer.chrome.com/docs/extensions/mv3/content_scripts/. (Online Accessed 3 January 2023).

Chrome Team, 2022a. Chrome DevTools. https://developer.chrome.com/docs/devtools/. (Online Accessed 29 May 2022).

Chrome Team, 2022b. Chrome extension API. https://developer.chrome.com/docs/extensions/reference/. (Online Accessed 28 May 2022).

Chrome Team, 2022c. tabCapture API. https://developer.chrome.com/docs/extensions/reference/tabCapture/. (Online Accessed 28 May 2022).

Chrome Team, 2023. Chrome DevTools protocol, log module. https://chromedevtools.github.io/devtools-protocol/tot/Log/. (Online Accessed 4 Jan 2023).

Deveria, A., Schoors, L., 2022. Browser support tables for content security policy (CSP) in web browsers. https://caniuse.com/?search=CSP. (Online Accessed 28 May 2022).

Filipe, R., Araujo, F., 2016. Client-side monitoring techniques for web sites. In: 2016 IEEE 15th International Symposium on Network Computing and Applications. NCA, pp. 363–366. http://dx.doi.org/10.1109/NCA.2016.7778642.

García, B., 2022. Hands-on Selenium WebDriver with Java. O'Reilly Media.

García, B., Gallego, M., Gortázar, F., Munoz-Organero, M., 2020. A survey of the selenium ecosystem. Electronics 9 (7), 1067.

García, B., Kloos, C.D., Alario-Hoyos, C., Munoz-Organero, M., 2022. Selenium-jupiter: A junit 5 extension for selenium WebDriver. J. Syst. Softw. 111298.

García, B., Munoz-Organero, M., Alario-Hoyos, C., Kloos, C.D., 2021. Automated driver management for selenium WebDriver. Empir. Softw. Eng. 26 (5), 1–51.

He, S., Zhu, J., He, P., Lyu, M.R., 2016. Experience report: System log analysis for anomaly detection. In: 2016 IEEE 27th International Symposium on Software Reliability Engineering. ISSRE, pp. 207–218. http://dx.doi.org/10.1109/ISSRE.2016.21.

Hossain, M., 2014. CORS in Action: Creating and Consuming Cross-Origin APIs. Simon and Schuster.

Hunt, J., 2019. Monkey patching and attribute lookup. In: A Beginners Guide To Python 3 Programming. Springer, pp. 325–336.

IEEE, 1990. IEEE standard glossary of software engineering terminology. In: IEEE Std 610.12-1990. IEEE, pp. 1–84. http://dx.doi.org/10.1109/IEEESTD.1990.101064.

Kalman, R.E., 1970. Lectures on Controllability and Observability. Tech. rep, Department of Operations Research, Stanford University, Stanford, CA.

Leotta, M., Clerissi, D., Ricca, F., Tonella, P., 2016a. Approaches and tools for automated end-to-end web testing. In: Memon, A. (Ed.), Adv. Comput. 101, 193–237. http://dx.doi.org/10.1016/bs.adcom.2015.11.007.

Leotta, M., García, B., Ricca, F., 2022. An empirical study to quantify the setup and maintenance benefits of adopting WebDriverManager. In: Vallecillo, A., Visser, J., Pérez-Castillo, R. (Eds.), Proceedings of 15th International Conference on the Quality of Information and Communications Technology. QUATIC 2022, In: CCIS, vol. 1621, Springer, pp. 31–45. http://dx.doi.org/10.1007/978-3-031-14179-9_3.

Leotta, M., García, B., Ricca, F., Whitehead, J., 2023. Challenges of end-to-end testing with selenium WebDriver and how to face them: A survey. In: Proceedings of 16th IEEE International Conference on Software Testing, Verification and Validation. ICST 2023, IEEE, p. (in press).

Leotta, M., Ricca, F., Tonella, P., 2021. SIDEREAL: Statistical adaptive generation of robust locators for web testing. In: Xie, T., Hierons, R.M. (Eds.), J. Softw. Test. Verif. Reliab. (STVR) 31, http://dx.doi.org/10.1002/stvr.1767.

Leotta, M., Stocco, A., Ricca, F., Tonella, P., 2016b. ROBULA+: An algorithm for generating robust XPath locators for web testing. In: Canfora, G., Dalcher, D., Raffo, D. (Eds.), J. Softw. Evol. Process (JSEP) 28 (3), 177–204. http://dx.doi.org/10.1002/smr.1771.

Mozilla MDN, 2022a. Browser extensions. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions. (Online Accessed 28 May 2022).

Mozilla MDN, 2022b. Firefox 101 release notes. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Releases/101. (Online Accessed 31 May 2022).

Nass, M., Alégroth, E., Feldt, R., Leotta, M., Ricca, F., 2023. Similarity-based web element localization for robust test automation. ACM Trans. Softw. Eng. Methodol. (TOSEM) (in press). http://dx.doi.org/10.1145/3571855.

Niedermaier, S., Koetter, F., Freymann, A., Wagner, S., 2019. On observability and monitoring of distributed systems–an industry interview study. In: International Conference on Service-Oriented Computing. Springer, pp. 36–52.

Ocariza, F., Bajaj, K., Pattabiraman, K., Mesbah, A., 2013. An empirical study of client-side JavaScript bugs. In: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 55–64. http://dx.doi.org/10.1109/ESEM.2013.18.

Ocariza, Jr., F.S., Pattabiraman, K., Zorn, B., 2011. JavaScript errors in the wild: An empirical study. In: 2011 IEEE 22nd International Symposium on Software Reliability Engineering. pp. 100–109. http://dx.doi.org/10.1109/ISSRE.2011.28.

Peña Moreno, N., 2022. Performance timeline level 2, W3C working draft. https://w3c.github.io/performance-timeline/. (Online Accessed 29 May 2022).

Quadri, S., Farooq, S.U., 2010. Software testing–goals, principles, and limitations. Int. J. Comput. Appl. 6 (9), 1.

Solntsev, A., 2019. How to get browser logs. https://selenide.org/2019/12/16/advent-calendar-browser-logs/. (Online Accessed 28 May 2022).

Stewart, S., Burns, D., 2022. WebDriver, W3C working draft. https://www.w3.org/TR/webdriver/. (Online Accessed 27 May 2022).

Yamanishi, K., Maruyama, Y., 2005. Dynamic syslog mining for network failure monitoring. In: Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining. KDD '05, Association for Computing Machinery, New York, NY, USA, pp. 499–508. http://dx.doi.org/10.1145/1081870.1081927.

Boni Garcia is an Associate Professor (with tenure) at Universidad Carlos III de Madrid in Spain. His main research interest is software engineering focusing on automated testing. He is a Staff Software Engineer at Sauce Labs in the Open Source Program Office. He is a committer at the Selenium project and creator of several projects belonging to its ecosystem, such as WebDriverManager, Selenium-Jupiter, or BrowserWatcher. He wrote the books Mastering Software Testing with JUnit 5 (Packt Publishing, 2017) and Hands-On Selenium WebDriver with Java (O'Reilly Media, 2022). He is the author of more than 45 research papers in different journals and conferences.

Filippo Ricca is an Associate Professor at the University of Genova, Italy. He received his PhD degree in Computer Science from the same University, in 2003, with the thesis "Analysis, Testing and Re-structuring of Web Applications". He is author (or coauthor) of more than 100 research papers published in international journals and conferences/workshops. In 2011 he was awarded the ICSE 2001 MIP (Most Influential Paper) award, for the paper: "Analysis and Testing of Web Applications" and in 2018 he was awarded the ICST MIP award. From 1999 to 2006, he worked with the Software Engineering group at ITC-irst (now FBK-irst), Trento, Italy. During this time he was part of the team that worked on Reverse engineering, Re-engineering and Software Testing. His current research interests include Web application testing and empirical studies in Software Engineering. The research is mainly conducted through empirical methods such as case studies, controlled experiments and surveys.

Jose M. del Alamo is currently an Associate Professor (with tenure) with the Universidad Politécnica de Madrid (DIT-UPM). His research interests include issues related to personal data management, including personal data disclosure, identity, privacy and trust management, and considering these aspects to advance software and systems engineering methodologies with a focus on their quality assessment. Dr. Del Alamo has been the Co-Chair of the IEEE International Workshop on Privacy Engineering, co-located to the IEEE Symposium on Security and Privacy, since 2015.

Maurizio Leotta is a Researcher at the University of Genova, Italy. He received his PhD degree in Computer Science from the same University, in 2015, with the thesis "Automated Web Testing: Analysis and Maintenance Effort Reduction". He is author or coauthor of more than 90 research papers published in international journals and conferences/workshops. His current research interests are in software engineering, with a particular focus on the following themes: Web, Mobile, and IoT application testing, functional test automation, empirical software engineering, business process modeling and model-driven software engineering.