

BACHELOR THESIS
COMPUTING SCIENCE



RADBOD UNIVERSITY

Design and implementation of a stealthy
OpenWPM web scraper

Author:

Daniel Goßen

s4751051

d.gossen@student.science.ru.nl

First supervisor/assessor:

Dr. Ir. Hugo Jonker

hugo.jonker@ou.nl

Second supervisor/assessor:

Dr. Ir. Erik Poll

E.Poll@cs.ru.nl

April 5, 2020

This page intentionally (almost) left blank

Abstract

OpenWPM [EN16] is a scraper framework designed to take privacy measurements on the web, for example to investigate online tracking. Jonker et al. [JKV19] identified that websites can detect web scrapers, and they found websites that behave differently to web bots than to regular visitors, e.g. by omitting advertisements or videos. Currently, it is not known in how far OpenWPM measurements are affected by web bot detection.

We analyze the stability of OpenWPM’s detectability to be a web bot, using the fingerprint surface of Jonker et al. [JKV19] and by applying JavaScript template attacks [SLG19]. Further, existing countermeasures such as user agent spoofing extensions and FP-Block are investigated for their effectiveness to reduce OpenWPM’s detectability.

Focusing on JavaScript spoofing, we derive the most advanced method to overwrite a revealing JavaScript object in order to minimize the risk of web bot detection. This results in a stealthy version of OpenWPM such that it’s fingerprint is much less distinguishable from a regular web browser. Further, it is shown common browser extensions spoofing JavaScript attributes are not robust against access inside (dynamically created) iframes.

With the stealthy version, the reliability of previous OpenWPM studies can be analyzed. Further, future studies can benefit from more accurate results.

Acknowledgment

Many thanks to my first supervisor Hugo Jonker for developing the idea of this thesis and the helpful support throughout the whole project. Especially our discussions helped to find the important essence of my experiments.

Further, thanks to Erik Poll for guiding me to Hugo Jonker as my first supervisor in the beginning and Benjamin Krumnow for the detailed explanations in the starting phase.

Finally, acknowledgments for the continuous support of my family and friends, particularly in times I had less motivation.

Contents

1. Introduction	1
2. Background	4
2.1. Browser fingerprinting	4
2.2. Web bots	5
2.3. Web bot detection	7
2.3.1. JavaScript template attack for web bot detection	7
2.3.2. Page deviations after detection	8
2.4. Design of OpenWPM	9
3. Related work	11
3.1. Browser fingerprinting	11
3.2. Detecting web bots	12
3.3. Fingerprinting prevention	13
3.4. Browser extensions and their detectability	14
4. How stable is OpenWPM's fingerprint surface?	15
4.1. Concept of stability	15
4.2. Measurement setup	16
4.2.1. Tweaks	17
4.2.2. Influence of full instrumentation	18
4.3. Analysis of stability	18
4.3.1. Different configurations and their fingerprint surface	18
4.3.2. Stability	21
4.3.3. Conclusion	21
5. Effectiveness of existing countermeasures	22
5.1. OpenWPM's build-in countermeasures	23
5.2. Effectiveness of typical Firefox extensions	25
5.2.1. The tool's concepts and functionality	25
5.2.2. <i>fingerprint2.js</i> analysis	26
5.2.3. Conclusion	28
5.3. State-of-the-art countermeasure: FP-Block 2.0	28

6. Detectability of different JavaScript spoofing methods	32
6.1. Methodology	32
6.2. About the <code>webdriver</code> attribute	33
6.3. Spoofing methods	34
6.4. Comparison of spoofing methods	35
6.5. Conclusions	38
7. Proof of concept implementation	39
7.1. The prototype	39
7.2. Validation experiment	41
7.3. Results	42
8. Future work	46
9. Conclusions	48
9.1. Results and answers to research questions	48
9.2. Impact of results	48
Bibliography	50
Appendices	55
A. Example fingerprint	56
B. Experiment environment	58
C. Firefox extensions: Page-, content- and background-scripts	59
D. Investigation of divergent window-sizes of OpenWPM	61
E. Spoofing detection in iframes	63
E.1. Results	63
F. Technical details on spoofing the <code>webdriver</code> property	67
G. Effects of <code>webdriver</code> detection	73

1. Introduction

Web scrapers are internet bots that automatically process data of websites, often on a large scale. Jonker et al. [JKV19] showed that web scrapers can be detected by websites, and that this detection may cause the site to send different content to the web scrapers than to regular users. In particular, parts of websites, such as advertisements or videos, were omitted in some cases.

The OpenWPM [EN16] scraper is a scraper framework designed to measure privacy, such as tracking used in many online advertisements. OpenWPM is available since 2014 and has been used in at least 54 studies¹. This shows the framework plays an important role in research. Currently, it is not clear to which extent the measurements taken by OpenWPM are affected by web bot detection.

Torres et al. [TJM15] introduced the fingerprint surface as the properties used to fingerprint users. During this thesis, we will also adopt this term to find and classify the differences between regular Firefox and OpenWPM by using their fingerprint surface based approach [JKV19]. Moreover, we apply JavaScript template attacks [SLG19] to extend the captured set of differences by a dynamic approach not depending on predefined attributes. While this measurement combination was used before [Kru+], we use it to determine the stability of identifying deviations across OpenWPM versions.

This work focuses on technical properties that differ, while not considering behavioral web bot detection. In particular, we will investigate the following main research question:

How can we build a stealthy version of the web scraper framework OpenWPM such that it is much less distinguishable from a regular web browser?

The main research questions leads to the following sub research questions:

- How stable are OpenWPM’s identifying properties across OpenWPM versions?
- What for countermeasures do already exist and what is their effectiveness?
- If existing countermeasures do not suffice, which method is suitable to reduce OpenWPM’s detectability?

To the best of our knowledge, no previous study examined improving OpenWPM’s stealthiness. But already as OpenWPM was published, the authors realized bot detection

¹<https://webtap.princeton.edu/software/> (last visited on 01/20/2020)

is a threat to the experimental results, so they added first countermeasures to the platform [Eng+15].

While answering the research question, we devise a stealthy version of OpenWPM which is much less detectable. The goal of this is to facilitate research into determining to what extent OpenWPM results are reliable. Further, future studies measuring privacy on the web could use the stealthier version to obtain more accurate results.

However, our goal is not a version of OpenWPM that is undetectable under all circumstances, but an improved version not detectable by common detection methods and frameworks. If parties especially target OpenWPM, detection will probably always remain possible.

Contributions

The main contributions of our work are:

- An analysis for the stability of OpenWPM’s detectable properties. Both the fingerprint surface by Jonker et al. [JKV19] and JavaScript template attacks [SLG19] are considered.
- An investigation of existing countermeasures and their effectiveness. This also reveals methods to improve state-of-the-art countermeasures such as FP-Block [TJM15; CY19].
- A comparison of the detectability of different JavaScript spoofing methods.
- A stealthier implementation of OpenWPM that reduces its detectability by overriding a revealing JavaScript object.²

Thesis structure

The thesis is organized as follows: [Chapter 2](#) presents the background about fingerprinting, web bots and OpenWPM. Then, [Chapter 3](#) discusses how our work compares to previous work in the literature. Next, [Chapter 4](#) measures the stability of OpenWPM’s fingerprint surface that allows for web bot detection, using the fingerprint surface based approach of Jonker et al. [JKV19] and JavaScript template attacks [SLG19]. In [Chapter 5](#), we analyze how effective OpenWPM’s own countermeasures, typical Firefox user agent spoofing extensions and FP-Block can address OpenWPM’s detectability. Further, [Chapter 6](#) focuses on spoofing on the level of JavaScript, using the example of OpenWPM’s `webdriver` attribute. Following, in [Chapter 7](#), we derive and validate a proof of concept implementation. Then, future work to be done in this area is given in [Chapter 8](#). Finally, [Chapter 9](#) summarizes our research and concludes this thesis.

²This lead to a pending contribution (pull request) for the official OpenWPM framework, see <https://github.com/mozilla/OpenWPM/pull/526> (last visited on 03/23/2020).

Availability

We publish our results and the stealthy version of OpenWPM.³

Ethical considerations

The question arises whether it is ethical to build a stealthy web scraper. As for this research, only the JavaScript `webdriver` property is spoofed. We therefore argue the risk of potential abuse is negligible when compared to the scientific interest.

However, this can be evaluated differently if a web scraper becomes almost undetectable for all types of detection. Then, it needs to be weighted up whether running such a scraper is accountable in the context of a particular study. It may be necessary to not disclose the source code and share the software only with bona fide researchers on explicit request.

³<https://github.com/Flnch/ba-thesis-resources>

2. Background

This chapter first discusses the general concept of browser fingerprinting. Then, it elaborates why web bots need to become more advanced in order to resemble regular users. This is followed by methods that can be used to detect web bots and common measures web servers take to counter web bots. Finally, the architectures of the OpenWPM scraper gets discussed. The specific architecture should be kept in mind to find the components that could contribute to OpenWPM detection.

2.1. Browser fingerprinting

Each human has a fingerprint that can re-identify him. In the context of web browsers, the fingerprint contains properties and behavior of:

- the system,
- the browser,
- and the user [Dol19].

The Screen resolution of a laptop, e.g. 1920×1080 , depicts a property of the system. Compared to the usually immutable fingerprint of a human, a browser fingerprint and its properties can change. Think of connecting your laptop to a video projector. At this moment, the screen resolution and thereby the browser's fingerprint changes.

Another example is canvas fingerprinting, where a specially crafted text is drawn in the browser. Since graphic drivers work differently and browsers do not implement canvas drawing in a standardized manner, the resulting image varies across devices. This behavior of the browser is not static, too. A browser update can for example change the canvas drawing engine.

An example of a collected fingerprint can be found in [A](#). While single attributes for themselves are seldom unique, the combination of enough attributes is it often well.

So why is fingerprinting used? In contrast to other techniques like cookies, fingerprinting is interesting because it is possible to re-identify a user on later visits without the need to store any information on the user side [Eck10].

On the one side, benign parties use fingerprinting for example to:

- Serve mobile phone users with a website version optimized for small screens;

- Support online authentication with fingerprinting as additional factor (if a user wants to log in from a unknown device, request another factor like e-mail confirmation for the login attempt);
- Prevent fraud such as bot generated fake accounts.

On the other side, fingerprinting can be used maliciously, for example as identifier to track a user across websites, link his surf behavior and show him targeted advertisements.

2.2. Web bots

Web bots are programs that visit websites for a special goal, without the need for direct user interaction. They can have several goals:

- The Googlebot visits internet websites and analyzes their content for keywords. Based on the keywords, it builds a web index that allows searching in the opposite direction: You enter a keyword and see websites that contain this keyword.
- The Linux program `wget` downloads content from a server.
- So called spambots collect publicly available e-mail addresses and send spam e-mails to them.

Using the de facto standard¹ `robots.txt`², websites can specify which parts of their website may be visited by web bots or not. While benign parties comply, actors can simply ignore the `robots.txt` settings of a website because they are not enforceable.

While they visit websites, most web bots should behave realistically, especially for web measurements. As modern web browsers constantly evolve, web bots need to go along and have to support current web technologies such as JavaScript or HTML5. This leads to web bots that get more close to actual browsers, for example PhantomJS³ (development mostly suspended since 2018), OpenWPM, Puppeteer or WebDriver.

This advanced web bots mostly rely on browser engines of regular web browsers. Their engines can run in two different modes, as depicted in [Figure 2.1a](#) and [Figure 2.1b](#):

1. **Headful (HF) browser mode:** This is the web browser version the reader commonly knows about. headful describes that there is a GUI and rendered content. Further, full functionality like JavaScript etc. is supported.

The regular Firefox or Chrome instances a user runs are examples of headful browsers.

¹While used for around 25 years, `robots.txt` is no official standard. But there is a recent proposal to the IETF, see <https://tools.ietf.org/html/draft-koster-rep-01> (last visited on 01/20/2020).

²<https://www.robotstxt.org/orig.html> (last visited on 01/20/2020)

³<https://phantomjs.org/> (last visited on 01/21/2020)

2. Background

2. **Headless (HL) browser mode:** In contrast, headless browsers do not implement all features of a headful browser. In particular, there is no GUI and content is not rendered on the screen. However, screenshots of the content can be exported.

This sort of browser is often used for automated testing during web development as it is said headless browsers save performance. It is also used if no screen hardware is available, for example on remote servers.

Chrome⁴ and Firefox⁵ both support a headless mode.

Further, it is possible to combine the advantages of both modes as shown in Figure 2.1c. OpenWPM supported this setup in the past: A underlying headful version of Firefox run in combination with the X virtual framebuffer (Xvfb) that replaces the actual screen hardware by a virtual screen. This way, full browser functionality (a headful browser) was combined with no screen output. However, this mode was removed from the project but due to detectability issues it may be re-implemented⁶.

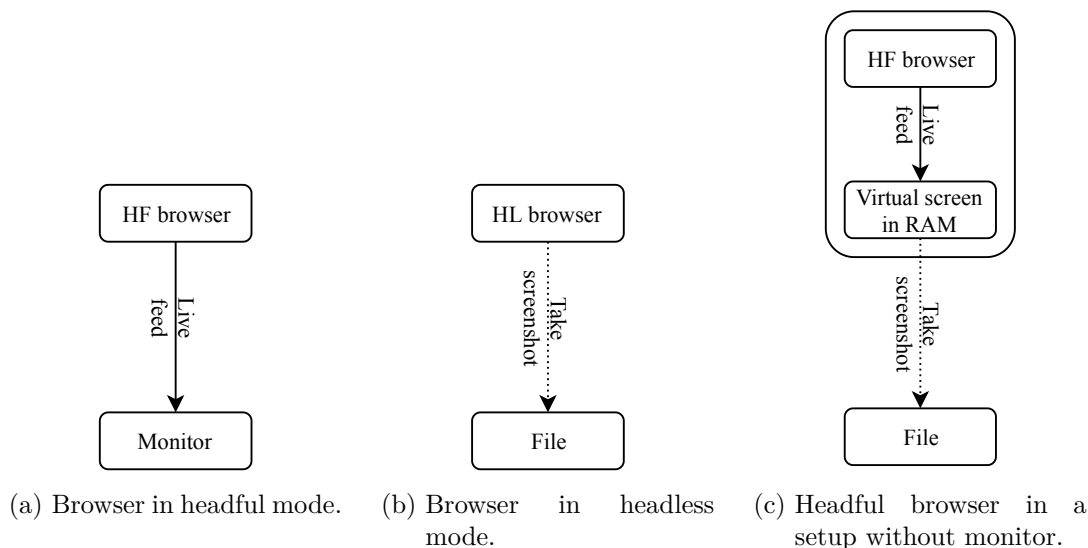


Figure 2.1.: Three different setups of web browsers.

⁴<https://chromium.googlesource.com/chromium/src+/lkgr/headless/README.md> (last visited on 01/20/2020)

⁵https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Headless_mode (last visited on 01/20/2020)

⁶<https://github.com/mozilla/OpenWPM/issues/504> (last visited on 01/20/2020)

2.3. Web bot detection

Using techniques such as fingerprinting, web servers can distinguish between regular users and web bots. Say a server receives a HTTP request with the user agent

```
Mozilla/5.0 (compatible; Googlebot/2.1;  
↪ +http://www.google.com/bot.html)
```

Then, the request likely originates from a web bot, namely the Googlebot (or someone impersonating the web bot). Behavior can also be evidence for a web bot [Chu+13], for example the typing speed.

Moreover, the detection of automated browsers is more difficult [JKV19] than the detection of simpler web bots such as `wget`. First, `wget` sends a revealing user agent like `Wget/1.13.4 (linux-gnu)`. Second, it is also detectable by the absence of JavaScript functionality.

In general, especially in headful mode, automated browsers behave more like regular users as they implement full browser functionality.

2.3.1. JavaScript template attack for web bot detection

Schwarz et al. [SLG19] developed a new method to find differences in browser environments, called JavaScript template attack. It finds even small and undocumented differences between JavaScript environments in an automated manner.

The main idea is to traverse the JavaScript prototype chain⁷. In JavaScript, it is not possible to receive all objects of the current environment at once. However, all objects are connected via the prototype chain that allow to access all their child and parent elements. This concept can be compared with inheritance of other object oriented programming languages like Java or C++. There is also the JavaScript DOM⁸ that specifies the hierarchy of objects, with `window` on top. Therefore, starting to traverse the prototype chain at `window`, it is possible to encounter all present properties.

Each run through the prototype chain is recorded in a template that afterwards can be compared with each other. If the environment changed across recordings, the differences in the templates indicate the actual differences between the browsers. So, for example comparing a template of Firefox with that of Chrome, amongst others reveals a difference in the `toString()` method:

⁷https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain (last visited on 01/21/2020)

⁸https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model (last visited on 01/21/2020)

2. Background

Browser	Value of <code>window.eval.toString()</code>
Firefox	<code>"function eval() {\n [native code]\n}"</code>
Chrome	<code>"function eval() { [native code] }"</code>

JavaScript template attacks can thus be used to compare regular browser environments with that of web bots / automated browsers. This reveals possible properties that allow to identify web bots.

However, currently there is one limitation of JavaScript template attacks. They only consider primitive types, that are numbers, strings and booleans. It is desirable to further include functions, but information about their semantic is needed in order to know how to call them correctly. Therefore, Schwarz et al. [SLG19] left this for future work. Only functions without parameters, e.g. `toString()`, can be added manually to the attack.

2.3.2. Page deviations after detection

Once a web bot is detected as such by a web server, the server can change its behavior against the web bot. This can result in a deviated web page with an observable difference [JKV19].

In general, server reactions can be classified as follows:

- **Same content:** The server responds with the regular content he would also serve to a human visitor.
- **Restrict access:** The client is asked to complete a CAPTCHA to proof he is human or he needs to log in with a user account. Otherwise, he will not see the actual web page.

This measure will hinder most web bots from seeing the actual content because it is either impossible to circumvent them or the effort is too extensive.

- **Alter/fake content:** While the web server altered it, for the client it will look as though he received the regular content. The web server may for example omit advertisements, multimedia content or login forms, or show a different graphic. Content may be faked completely when an entirely different site is served.

Generally, search engine crawlers such as the Googlebot punish maliciously deviated websites, as this technique is used to improve search rankings.

Client-side, altered content is the most difficult to detect. Further, in the context of web measurement studies, they may lead to tampered results if the difference is not observed by the researchers.

- **Block request:** The server either does not respond at all or he responds with an “access denied” message.

While the client does not receive the actual web page content, he can easily detect the server refused him access.

2.4. Design of OpenWPM

OpenWPM is designed to perform large-scale measurements on the web by automated scraping of websites [EN16]. To reach this goal, several components are put together that can be assigned to different layers. A high level overview of the architectural design and the interaction of OpenWPM’s different components is depicted in Figure 2.2 and will be discussed in the following.

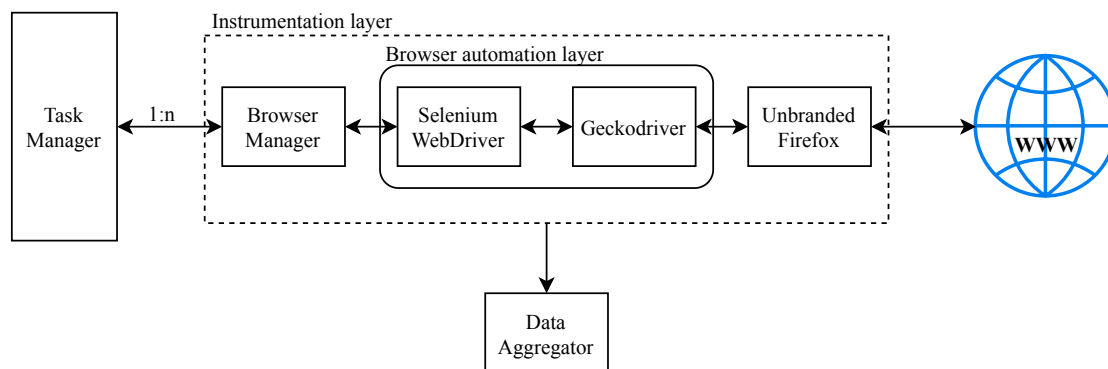


Figure 2.2.: Architecture of OpenWPM, modification of [EN16; Kru+].

On top, the task manager is the controlling instance that manages parallel scraping. One single task manager instance is responsible for all browser instances. It controls the instrumentation layer that, in the end, is responsible to receive and record all data encountered during website scraping. Like all “glue” components of OpenWPM, the task manager is written in Python 3.

For each desired browser instance, the task manager starts a browser manager that basically calls the next module and sets some settings. The browser manager is the bridge between the task manager and browser driver. OpenWPM uses Selenium WebDriver as browser driver.

W3C WebDriver⁹, or simply WebDriver, is a control interface to interact with browsers in an automated manner. It is especially used for automated browser-testing and all common browsers support it.

⁹<https://www.w3.org/TR/webdriver/> (last visited on 01/21/2020)

2. Background

OpenWPM uses Selenium WebDriver¹⁰, that, on the other hand, is one particular W3C WebDriver client. It is compatible with a variety of browsers such as Firefox and Chrome. This opens the possibility to change the specific browser that OpenWPM uses. However, OpenWPM officially only supports Firefox.

Selenium WebDriver, together with Geckodriver, forms the browser automation layer. This layer is responsible for issuing commands to the web browser. While Selenium WebDriver provides a Python 3 API to control the web browser, geckodriver (not to be confused with Gecko) is a proxy in front of Firefox. It translates calls using the W3C WebDriver protocol into calls of the Marionette protocol because instrumenting Firefox is only possible using the Marionette protocol.

Next, the web browser Firefox receives the instrumentation commands and interacts with the actual web pages. Not a regular version of Firefox but Firefox unbranded¹¹ (a form of Firefox Nightly) is used because OpenWPM includes unsigned Firefox extensions (not to be confused with Firefox plugins [SN17]). A regular Firefox would refuse to install unsigned extensions but the unbranded version allows to disable this signature check.

Finally, the data aggregator receives and stores the data of all instrumentation components in a central log on disk.

¹⁰<https://selenium.dev/documentation/en/webdriver/> (last visited on 01/21/2020)

¹¹https://wiki.mozilla.org/Add-ons/Extension_Signing#Unbranded_Builds (last visited on 01/21/2020)

3. Related work

3.1. Browser fingerprinting

In the last decade, since Eckersley’s study [Eck10] introduced the term “browser fingerprinting” as re-identifying browsers using their own properties, extensive research is done for the impact of browser fingerprinting and techniques how to circumvent it. Over time, researchers constantly identified new and more advanced techniques for identification (e.g. [MS12; Mul+13; Ung+13; FE15; EN16; Ole+16; SN17; Sta+19]). In their survey study, Laperdrix et al. [Lap+19] give an extensive overview of the whole research area.

As browser fingerprinting is conducted on a wide scale, researchers try to scale up their measurements using automated or semi-automated approaches. Therefore, Englehardt and Narayanan [EN16] for example introduced their framework OpenWPM to perform automated large-scale privacy measurements on websites. It allows visiting websites and analyzing their behavior, for instance whether they perform a certain fingerprinting technique, on a large scale. OpenWPM follows a modular design. It uses the browser automation tool Selenium that serves a high level programming interface such that the integrated Firefox browser can be exchanged. Section 2.4 discusses the design in more detail. Because of the abstract design and OpenWPM’s active maintenance (since April 2019 by Mozilla), OpenWPM is used in at least 54 studies.¹ In their study, Englehardt and Narayanan point to the problem that web bot detection can possibly influence results, but they reserve further countermeasures besides basic ones for future work [Eng+15].

FPDetective by Acar et al. [Aca+13] is a similar framework. However, in comparison, OpenWPM has the advantage that its instrumentation is more generic and does not require modifications of the browser’s source code. Especially maintaining customized browser source code is time expensive.²

¹<https://webtap.princeton.edu/software/> (last visited on 01/20/2020)

²FPDetective was last modified in December 2015, see <https://github.com/fpdetective/fpdetective> (last visited on 04/04/2020).

3.2. Detecting web bots

Web bots are detected on different layers that can be divided into behavior- and property-based detection [Dol19]. A property could be a version number, while the way the hardware renders a text glyph is an example for behavior (see Section 2.1 for detailed examples). This thesis only considers property-based detection to limit its scope.

Vlot [Vlo18] and, directly following from Vlot’s work, Jonker et al. [JKV19] focus on the fingerprint surface to measure web bot detection. Therefore, they compare different web bot fingerprints with that of regular browsers to determine the properties revealing web bots.

Their analysis of the Alexa top 1 million shows that 12.8% of the scanned websites are likely to use web bot detection. In particular, Jonker et al. found websites omitting certain content such as advertisements or videos if accessed by a web bot. One encountered detection attribute is the JavaScript `webdriver` property which is spoofed in our stealth OpenWPM prototype.

Further, we use their fingerprint surface measurement tool to inspect deviations caused by OpenWPM (Chapter 4) and to inspect the detectability of different JavaScript spoofing methods (Chapter 6). In addition to attributes derived by reverse analyzing commercial fingerprinters, their measurement tool integrates *fingerprint2.js*³, an widely used open source JavaScript framework for fingerprint generation that also contains various fingerprint integrity checks.

Schwarz et al. [SLG19] propose their JavaScript template attacks to automatically find divergent browser properties across different browser environments. Their method traverses the whole JavaScript prototype chain. Hence, it considers all attributes accessible via JavaScript. Details of the technique are explained in Subsection 2.3.1. We use template attacks to compare the environments of a regular Firefox instance with that of OpenWPM, as first done by Krumnow et al. [Kru+].

Tailored to OpenWPM, Krumnow et al. [Kru+] investigated in how far the framework is detectable via fingerprinting and which components cause the detectability. They consider both detection of OpenWPM as itself and as a web bot in general, using the measurement frameworks by Jonker et al. [JKV19] and Schwarz et al. [SLG19]. Their results show that OpenWPM headful is mainly detectable via screen resolution and the `navigator.webdriver` DOM property, while OpenWPM headless offers a wider detection surface. The detectability also grew significantly for both the headful and headless mode while OpenWPM’s JavaScript instrumentation was activated.

³<https://github.com/Valve/fingerprintjs2/tree/437c9636bede60e6441ee22445434067c8db93fe>
(last visited on 04/04/2020)

Moreover, Krumnow et al. propose a detector specifically identifying OpenWPM as OpenWPM (and not as a web bot in general). This detector is not applicable to our research as we focus on OpenWPM being detected to be a web bot. In that sense, the detector by Krumnow et al. is too severe as it only triggers for OpenWPM instances but not if only indicators of a web bot are present.

3.3. Fingerprinting prevention

As fingerprinting turned out to be an effective and in-the-wild used technique to re-identify users on the internet, researchers started investigating how to prevent or at least weaken the effectiveness of fingerprinting.

Nikiforakis et al. [NJJ15] propose PriVaricator that introduces randomness to the browser's fingerprint by spoofing fingerprint related browser properties. Therefore, they modified the browser's source code because the approach of a JavaScript extension did not work for both Firefox and Chrome.

A similar approach is taken by Laperdrix et al. [LBM17] who built a modified version of Firefox called FPRandom. It also alters the browser's source code.

Otherwise, FaizKhademi et al. [FZW15] developed FP-Guard that modifies fingerprinting attributes by a combination of a browser extension and browser source code modification.

Yet another approach is used in Blink by Laperdrix et al. [LRB15]. Blink uses virtualization to generate a random browsing environment consisting of operating system, browser, installed plugins etc. Therefore, this approach does not introduce inconsistencies in the fingerprint as the fingerprint itself is not spoofed. However, this solution certainly adds performance overhead.

Finally, Torres et al. [TJM15] introduce the Firefox extension FP-Block to prevent fingerprint-based tracking. Compared to the previous solutions, FP-Block specifically prevents tracking across websites while still allowing regular tracking. They distinguish two forms of tracking: On the one hand, *regular tracking* only takes place inside the tracker's own website, while, on the other hand, *cross-domain tracking* also tracks across different websites.

The authors argue that blocking regular tracking inside one page frequently breaks functionality. They further state pure cross-site fingerprinting prevention reduces the amount of privacy invading tracking to an acceptable minimum while maintaining page functionality. This design is well suited for OpenWPM because its large-scale automated

crawls should not be traceable across websites.

In 2019, Cosijn and Yasko [CY19] released an improved version FP-Block 2.0. The extension’s speed was enhanced by a new way of fingerprint generation that generates all fingerprints at the first start of the extension. Further, the previous FP-Block version introduced inconsistencies to the browser fingerprint. Cosijn and Yasko resolved bugs in object, canvas and font fingerprinting along with conflicting properties inside generated fingerprints.

Therefore, FP-Block is a comprehensive and complete state-of-the-art academic tool to counter cross-site tracking. The generated fingerprint profiles are constructed such that they form a coherent set of spoofed values. Equally important, the set of spoofed attributes contains all major attributes used by in-the-wild fingerprinters.

3.4. Browser extensions and their detectability

Several academic and non-academic tools are developed as browser extensions to add functionality to the browser. Examples include fingerprinting countermeasures, providing a dark mode for websites or user agent spoofing. However, researchers found websites can detect presence of many extensions as they introduce observable differences.

Nikiforakis et al. [Nik+13] analyzed user agent spoofing extensions and found some are detectable by an incomplete coverage of spoofed attributes or incoherent values leading to impossible combinations. Further, they found the property order of DOM objects such as `navigator` to be different between browser versions. Therefore, it is in principle necessary to also spoof this enumeration behavior.

In a different setting, Storey et al. [Sto+17] hide DOM modifications by their ad blocker by overwriting the global JavaScript `toString` function.

Starov and Nikiforakis [SN17] found that 9% of the sample of 10,000 browser extensions can be detected as they leak information to the browser’s DOM. In a more recent work, Starov et al. [Sta+19] analyzed Google Chrome extensions for “extension bloat”, that are omissible page modifications not adding functionality. Extension bloat was present in 5.7% of the tested extensions and increased the ability to fingerprint users.

Vastel et al. [Vas+18] used their prototype FP-Scanner to analyze state-of-the-art techniques to counter fingerprinting and to which extent they introduce fingerprint inconsistencies that enable detection. One detection feature is again the enumeration order of the `navigator` object. However, they argue detecting that a fingerprinting countermeasure is applied does not necessarily improve the fingerprinting ability. This depends on the type of leaked information.

4. How stable is OpenWPM’s fingerprint surface?

Previous work measuring OpenWPM’s fingerprint surface, such as [JKV19] or [Kru+], tested specific versions of OpenWPM and its corresponding software components. However, functionality and behavior changes during OpenWPM’s evolution and so can the previously identified fingerprint surface of OpenWPM. A changed fingerprint surface can, for example, render existing detection useless or introduce new identifying attributes. Krumnow et al. [Kru+] looked into the stability of divergent properties across different GeckoDriver versions.

In this chapter, we extend their findings by repeating their measurements with a recent version of OpenWPM showing the main findings are stable across major OpenWPM versions. Thereby, we also check for potential new identifying deviations. The order of attributes inside HTTP headers is newly found to be identifying, although comparison with Krumnow et al. shows it is not stable across OpenWPM versions. Further, we introduce a concept of stability with different stability levels to describe how stable a fingerprint surface is.

4.1. Concept of stability

Before analyzing, we first define the concept of stability. In more detail, we investigate the stability of the fingerprint surface identifying OpenWPM to be a web bot.

When speaking of stability, something can only be considered stable or unstable in relation to a baseline. In our case, the previous findings of Krumnow et al. [Kru+] form the baseline of OpenWPM’s fingerprint surface and our repeated measurements are compared against those. That way, two different major versions of OpenWPM are compared against each other.

For this comparison, and thus for testing stability, it is necessary to know technical properties about both versions. This information is available by the fingerprint surfaces captured by the measurement frameworks.

If we now compare two fingerprint surfaces on property level, different scenarios regarding stability are possible:

A fingerprint surface property with the same value for both measurements. In this case, the property is *stable*. As an example, an (identifying) user agent string that stays

4. How stable is OpenWPM’s fingerprint surface?

the same would be stable.

A fingerprint surface property with a (partly) changed value between both measurements. Here, two cases need to be distinguished:

1. If the different part is not essential for web bot detection, the property is still considered *stable*. An (identifying) user agent string containing another version number but otherwise staying the same would match this category.
2. If the different part is essential for web bot detection, two further sub-cases exist:
 - In case both different values allow web bot detection, it is still considered *restricted stable*. The attribute remains usable for web bot detection but only in a restricted form as detecting parties would need to adopt their detection criteria. An example of such a case would be a user agent string containing the keyword “automated” instead of “web bot”.
 - In case the deviation removes the ability for web bot detection, we have a *unstable* property, e.g. when a user agent string removed the keyword “automated”, or a identifying property is removed completely.

4.2. Measurement setup

This section describes our setup and necessary tweaks of the hybrid measurement approach by Krumnow et al. used to repeat their measurements. The approach uses both the fingerprint surface measurement framework by Jonker et al. [JKV19] and JavaScript template attacks [SLG19] to find the technical properties that are different from that of regular Firefox.

The measurements are taken on a laptop with a new Ubuntu installation¹. Both measurement frameworks serve a local web page to capture configurations. Some measured properties are hardware dependent (e.g. vendor, system architecture etc.) and not all our results are stable across different machines, but this chapter leverages evidence that all main identifying properties are stable.

OpenWPM can run in headless and headful mode, each with different activated mechanisms that log parts of the crawling process. JavaScript instrumentation for example logs JavaScript calls and access to some fingerprint related properties. Therefore, to find the divergent technical properties and their origin, we visit both test pages with different configurations.

From Krumnow et al. [Kru+] it is already known that JavaScript instrumentation is the only form of instrumentation contributing to OpenWPM’s fingerprint surface.

¹For details about the configuration, see [Appendix B](#).

Therefore, we do not test each instrumentation on its own but only JavaScript instrumentation and all instrumentation components at once. Altogether, this results in the following test configurations:

- **❶** Regular Firefox (the baseline)
- OpenWPM headful
 - **❷** Without instrumentation
 - **❸** Only with JavaScript instrumentation
 - **❹** With full instrumentation
- OpenWPM headless
 - **❺** Without instrumentation
 - **❻** Only with JavaScript instrumentation
 - **❼** With full instrumentation

With the captured snapshots, we can compare configurations, especially the baseline (**❶**), against others.

4.2.1. Tweaks

Performing the measurements requires some tweaks of the affected components.

First, OpenWPM needs to be instructed to automatically enter information to the measurement frameworks (such as typing the configuration name or clicking a button) because the measurement frameworks are not designed for automation².

Second, the Firefox version integrated in OpenWPM is only occasionally updated and lacks behind. When comparing OpenWPM to regular Firefox, only differences introduced by OpenWPM should be measured and no additional differences caused by another Firefox version. Therefore, the same older Firefox version is taken as reference. However, automatic updates of Firefox need to be disabled explicitly. Other than that, the regular Firefox runs in standard settings.

Third, performing JavaScript template attacks for configuration **❸**, **❹**, **❻** and **❼** at first triggered OpenWPM timeouts because the various instrumentation components slowed down the attack. It turned out the OpenWPM `get` command timed out although the `get` function was already finished at the time the actual template attack begun. Increasing the `get` command's timeout fixed the problem.

²We reused code from StealthBot [JKV19] (<https://github.com/bkrumnow/StealthBot/tree/5f125c16ec9b9b17cefb1e90552fb49e88d658f1> (last visited on 02/07/2020)) to perform the necessary automation steps.

4.2.2. Influence of full instrumentation

Now, it is examined whether full instrumentation, as expected because of [Kru+], does not lead to further detectable differences compared to exclusive JavaScript instrumentation. Otherwise, we would need to consider each instrumentation component on its own.

Therefore, both the template attack and fingerprint surface measurements of full instrumentation are compared to those with only JavaScript instrumentation, that is:

- ④ ⇔ ③ for both measurement methods
- ⑦ ⇔ ⑥ for both measurement methods.

The comparison shows that full OpenWPM instrumentation does not add more detectable differences than OpenWPM with JavaScript instrumentation already does. Hence, we can conclude the following forms of instrumentation do not introduce detectable differences:

- HTTP instrumentation
- cookie instrumentation
- navigation instrumentation
- storage of all response bodies.

This confirms our expectations such that configuration ④ and ⑦ do not need to be considered anymore for further analysis.

4.3. Analysis of stability

To analyze the stability of OpenWPM's fingerprint surface regarding web bot detection, our results are compared to that of Krumnow et al. Both our results and the comparison is depicted in Table 4.2, which will be discussed in the following.

4.3.1. Different configurations and their fingerprint surface

Table 4.2 shows our measurements of the four relevant OpenWPM configurations, along the “expected” column that represent the values captured for a stand-alone Firefox instance. The columns containing OpenWPM configurations only include a value if it deviates from the baseline (stand-alone Firefox). In other words, the cells with content represent OpenWPM's fingerprint surface for web bot detection. For now, the coloring is ignored until later.

Regarding the extent of deviations, the findings of Krumnow et al. [Kru+] are confirmed. In short, on the one hand, OpenWPM’s JavaScript instrumentation introduces a handful of deviations on the JavaScript level. On the other hand, OpenWPM’s headless mode causes several deviations such as missing WebGL functionality or different window-sizes. Details can be extracted from Table 4.2 or found in [Kru+].

However, we measured one new feature for OpenWPM detection: The position of the `referer` attribute inside HTTP request headers is different, as depicted in Figure 4.1. Firefox sends the `referer` field always after the `connection` field, while these two fields are reversed for OpenWPM. RFC 7230 states the field order is not important³, but it could nevertheless be used for detection. The corresponding method is called HTTP fingerprinting. This deviation was present in all four configurations. Therefore, it cannot be caused by OpenWPM’s instrumentation or the headless/headful modes. Inspecting OpenWPM’s integrated Firefox and Geckodriver by running it stand-alone via Selenium shows that the deviation is not present there. Thus, it needs to be caused by some main OpenWPM Python component. Unfortunately, we were unable to locate the exact source.

```

▼ Hypertext Transfer Protocol
  ▶ GET /stat?c1=RmlyZWZveF820S4wLjFfVWJ1bnR1XzE4LjA0LjNfTFRT&c2=RmlyZWZveF820S4wLjFfVWJ1
    Host: localhost:8000\r\n
    User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:70.0) Gecko/20100101 Firefox/70.0\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
    Accept-Language: en-US,en;q=0.5\r\n
    Accept-Encoding: gzip, deflate\r\n
    Connection: keep-alive\r\n
    Referer: http://localhost:8000/\r\n
    Upgrade-Insecure-Requests: 1\r\n
    \r\n

```

(a) HTTP request header of a *Firefox* instance.

```

▼ Hypertext Transfer Protocol
  ▶ GET /stat?c1=RmlyZWZveF820S4wLjFfVWJ1bnR1XzE4LjA0LjNfTFRT&c2=RmlyZWZveF820S4wLjFfVWJ1
    Host: localhost:8000\r\n
    User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:70.0) Gecko/20100101 Firefox/70.0\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
    Accept-Language: en-US,en;q=0.5\r\n
    Accept-Encoding: gzip, deflate\r\n
    Referer: http://localhost:8000/\r\n
    Connection: keep-alive\r\n
    Upgrade-Insecure-Requests: 1\r\n
    \r\n

```

(b) HTTP request header of an *OpenWPM* instance.

Figure 4.1.: Comparison of HTTP request headers between Firefox and OpenWPM.

³<https://tools.ietf.org/html/rfc7230#section-3.2.2> (last visited on 02/12/2020)

4. How stable is OpenWPM’s fingerprint surface?

Table 4.2.: Features of four different captured OpenWPM configurations, each compared to regular Firefox. Only *deviations* with respect to the expected value (regular Firefox) are given. The *cell color* depicts the *stability* of this finding in comparison to [Kru+]. The measurement method for each feature is given in the left most column.

Method	Feature	Expected	OpenWPM			
			Headful (HF)		Headless (HL)	
			② No JS instr.	③ JS instr.	⑤ No JS instr.	⑥ JS instr.
Both	<code>navigator.webdriver</code>	false	true	true	true	true
	<code>window.jsInstruments</code>	undefined		defined		defined
	<code>window.instrumentFingerprintingApis</code>	undefined		defined		defined
	Available resolution ²	1299x741			1366x768	1366x768
	WebGL support ²	yes			no	no
FP-surface ¹	HTTP request header: Position of <code>referer</code> field in relation to <code>connection</code> field	after	before	before	before	before
Template attacks	<code>window.j</code>	defined			undefined	undefined
	Nr. of new objects on <code>navigator.languages.__proto__</code>	0			43	43
	Nr. of native JS functions overwritten with logging	0		222 ³		222 ³
	Nr. of divergent <code>canvas</code> window-size prop.	0	14	14	14	14
	Nr. of divergent <code>window</code> window-size prop.	0	8 ⁴	8 ⁴	14 ⁴	14 ⁴
	Nr. of removed <code>window.wgl</code> objects	0			1182 ⁵	1182 ⁵
	Nr. of removed <code>window.gl</code> objects	0			892	892

- *Stable* fingerprint surface property
- *Restricted stable* fingerprint surface property
- *Unstable* fingerprint surface property

¹ Using the fingerprint surface based framework BrowserBasedBotFP by Jonker et al. [JKV19].

² Template attacks measure this more detailed on the JavaScript level, see the corresponding rows.

³ All on the objects `window.canvas`, `window.sessionStorage` and `window.localStorage`.

⁴ Krumnow et al. [Kru+] additionally measured a deviation of `window.screen.height` and `window.screen.width`.

⁵ Krumnow et al. [Kru+] measured 13 additionally removed `window.wgl` objects.

4.3.2. Stability

The coloring of the cells indicates the stability of each finding compared to Krumnow et al. [Kru+]⁴, where green corresponds to *stable*, orange to *restricted stable* and red to *unstable* fingerprint surface properties according to our concept of stability (see Section 4.1). There are also further new and removed JavaScript properties and functions. However, they are caused by Firefox updates and thus unrelated to OpenWPM detection. They are not considered in this analysis.

From Table 4.2, it can be seen that most findings, such as the `webdriver` attribute or overwritten functions for logging, remain stable. They are thus suited to detect OpenWPM and will likely remain stable in the future.

Some window-size related properties are only *restricted stable* because they (can) differ from machine to machine, but on the same machine they can be used to tell regular Firefox and OpenWPM apart. Therefore, the window-size can only be used as criteria for detecting OpenWPM if additional information about the client machine is known or derived by another method.

Finally, the HTTP request header position is considered unstable because it was only measured in our data but not in that of Krumnow et al. [Kru+]. Our tested version of OpenWPM can be identified by this deviation, while it remains unclear whether this will remain possible in the future.

4.3.3. Conclusion

All in all, the comparison of our repeated measurements with the findings of Krumnow et al. [Kru+] shows the main features for OpenWPM detection can be considered stable. Unfortunately, the newly identified revealing position of the HTTP `referer` attribute needs to be considered unstable for the time being. Future work could repeat this measurement for one of the following major OpenWPM versions to further increase the confidence in stability.

⁴Krumnow et al. used OpenWPM with Firefox 68.0 and Geckodriver 0.24.0, while we used OpenWPM with Firefox 70.0 and Geckodriver 0.24.0.

5. Effectiveness of existing countermeasures

As shown in [Chapter 4](#), it is possible to detect OpenWPM on basis of technical properties. This chapter investigates whether there already exists an easy solution to address OpenWPM’s detection. We distinguish three different levels of countermeasures:

1. Build-in countermeasures;
2. Typical browser-extension countermeasures;
3. State-of-the-art countermeasures.

First, on the level of OpenWPM, its own countermeasures are analyzed. Their functionality is already present in the program and thus can be used without additional effort.

Second, we picked five of the most popular Firefox user agent spoofing extensions. They change the browser’s fingerprint and are therefore analyzed using *fingerprint2.js*. *fingerprint2.js* is the de facto open source fingerprinting utility. Here, we use it to check for the integrity of the spoofed fingerprints.

Third, we investigate on the level of advanced anti cross-site tracking solutions. We examine FP-Block 2.0 that, as already mentioned in [Related work](#), depicts a state-of-the-art academic measure against cross-site tracking. *fingerprint2.js* is similarly used to check for fingerprint integrity. Moreover, the fingerprint surface based measurement framework by Jonker et al. [[JKV19](#)] is used to capture deviations in the fingerprint surface when FP-Block is added to the browser. This is completed by a JavaScript template attack that works more abstract.

Testing countermeasures on each of these three levels shows they all have a positive effect on OpenWPM’s detectability, but none provides the perfect, complete solution. Even FP-Block 2.0, that is very effective to counter cross-site tracking but not hardened against anti-tracking detectors, can be identified by websites.

Therefore, we conclude that a combined solution is necessary: FP-Block’s advanced cross-site tracking countermeasures need to be combined with stealthy JavaScript spoofing mechanisms. Such a solution is capable of defeating cross-site tracking while at the same time not handling suspiciously on the level of JavaScript. To make the next step, [Chapter 6](#) analyzes JavaScript spoofing methods for the most handy one.

5.1. OpenWPM's build-in countermeasures

Currently, OpenWPM offers two measures against web bot detection:

1. Perform random actions on websites;
2. Start the browser with randomized attributes.

On the one hand, the first measure performs random mouse movements, random scrolling and adds random delays between commands issued to OpenWPM. Such commands include receiving an actual web page or typing in user credentials to a page. This basic countermeasure addresses behavioral bot detection, so it does not help in our case of property based detection.

On the other hand, the second countermeasure starts each browser instance with a screen resolution and user agent randomly taken from predefined lists. Especially defining the screen resolution is interesting, as [Chapter 4](#) revealed that OpenWPM's resolution differs from that of stand-alone Firefox. The second countermeasure will be investigated after the next paragraph.

Further, Firefox has a built-in tracking protection¹ that blocks requests to URLs Mozilla considers doing cross-site tracking. Unfortunately, this solution is currently not usable in OpenWPM². Anyway, it is questionable whether this measure would be useful for OpenWPM crawls as it is not under the researchers control which URLs Mozilla considers to be doing cross-site tracking.

We now turn to OpenWPM's random attribute countermeasure in more detail. Actually, it consists of two different measures.³

The first allows to manually overwrite the user agent string. Internally, this is done by setting the Firefox configuration preference `general.useragent.override`. While this perfectly allows to spoof the user agent, it is a special option provided by Firefox only for a few fingerprint-related properties. Therefore, it is unfortunately not possible to spoof arbitrary properties such as `navigator.webdriver` via the same method.

The second feature allows specification of (a set of) screen resolutions. Therefore, one sets the `random_attributes` option and specifies a list of resolutions⁴. This possibility is interesting as OpenWPM's resolution differs from that of regular Firefox (see [Chapter 4](#)). An analysis in [Appendix D](#) shows that the differences are caused by the operating

¹https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Privacy/Tracking_Protection (last visited on 02/07/2020)

²There are issues with Firefox's telemetry service, see <https://github.com/mozilla/OpenWPM/issues/101> (last visited on 02/07/2020).

³It should be noted that the random resolution option is coupled with the user agent spoofing feature. Alternatively, if a single static spoofed resolution is sufficient, one can set `DEFAULT_SCREEN_RES` in `deploy_firefox.py` to the desired value.

⁴To be specified in `screen_resolutions.txt`.

5. Effectiveness of existing countermeasures

system's skin, shadow effects etc. that behave differently for maximized and normal windows, together with divergent sizes of newly spawned windows.

It is important to know the way OpenWPM defines the window size. A specification inside `screen_resolutions.txt` internally defines `'screen_res'`. However, this term is somewhat misleading, as it actually sets the size of OpenWPM's browser window not including the window titlebar by the operating system. Figure 5.1 shows an example. Further, if the defined resolution is greater than the usable free place on the display, a window of the maximum possible size is spawned. Thus, under operating systems such as Ubuntu that have a permanent side and top bar, the possible area for program windows is smaller than the hardware screen resolution.

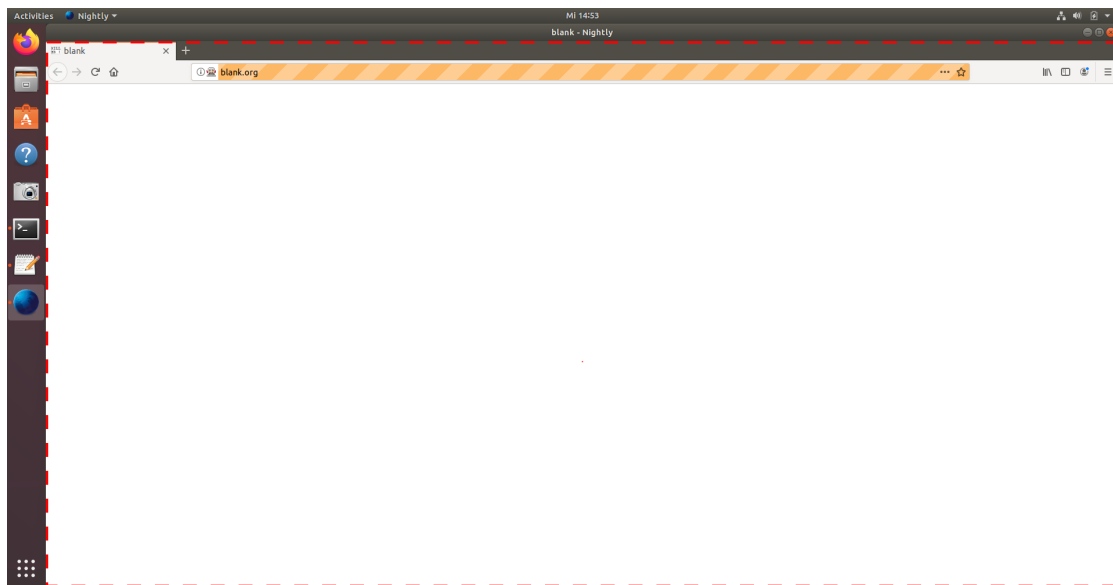


Figure 5.1.: An OpenWPM instance where the area defined by OpenWPM's `'screen_res'` setting is highlighted as a red box. In this example, the specified resolution is 4000x3000, but the available resolution is smaller such that OpenWPM only uses a `'screen_res'` area of 1849x940. In terms of JavaScript, the highlighted area has the dimensions `window.outerWidth` x `window.outerHeight`.

In general, all decorations and skin differences of the window manager influence the available area for windows: The existence and position of a title bar, side bar, window borders, a status bar, vertical and horizontal scroll bars, task bar etc. All mainstream operating systems at least contain bars of some type. Therefore, we argue it is suspicious if the browser viewport of a desktop machine contains a common full screen display resolution such as 1920x1080 or 1366x768. Thus, to make OpenWPM more stealthy, we

propose to choose a common resolution and subtract some pixels in x and/or y direction before using it for OpenWPM's '`screen_res`'. Then, all OpenWPM windows are of this specified size and cannot be told apart from regular Firefox instances.

So, while being useful to influence OpenWPM's window-size, the built-in countermeasures for web bot detection are not sufficient to hide OpenWPM's fingerprint surface beyond that. However, specifying OpenWPM's window-size already improves the situation.

5.2. Effectiveness of typical Firefox extensions

To test standard Firefox extensions for their spoofing effectiveness, we choose five of the most popular Firefox extensions from Mozilla's extensions archive that spoof the browser's user agent. They are used by 8,700 up to 95,800 users. In the following, we introduce the concept of each extension, compare their functionality and test them with *fingerprint2.js*.

5.2.1. The tool's concepts and functionality

Table 5.2 gives an overview of the chosen extensions.

Three extensions use the concept of randomizing the user agent. They allow to either choose a new random user agent each time the browser is started or to change it in a fix interval. Compared to special tools such as FP-Block, the tested extensions only use a simple concept of randomization. FP-Block for example separates different fingerprint profiles for each visited domain.

Next, all tested extensions spoof the user agent both inside the HTTP header and inside the JavaScript `window.navigator` property. Only spoofing at one place would immediately disclose the true user agent.

For a coherent fingerprint not only the user agent plays a role. Unfortunately, *Random User-Agent* and *User-Agent Switcher by Linder* only spoof the user agent. Fortunately, the remaining three extensions also spoof additional JavaScript properties such as `navigator.platform`. This is important, as a user agent spoofing a Windows computer combined with a Linux platform makes no sense and is suspicious.

Some extensions also contain further, advanced functionality. *Chameleon* for example can spoof more HTTP header fields such as the source `referer` or it can prevent ETag⁵ tracking. However, here we investigate the standard features.

Finally, for *Chameleon*, two modes exist: A basic mode with less spoofing functionality and a mode with active script injection. The latter injects a JavaScript content script

⁵<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/ETag> (last visited on 03/26/2020)

5. Effectiveness of existing countermeasures

Table 5.2.: Comparison of five popular user agent spoofing extensions from the Firefox extension archive.

Information/functionality	Firefox spoofing extension				
	Chameleon ¹	Random User-Agent ²	UAS by Linder ³	UAS by Alexander Schlarb ⁴	UAS and Manager ⁵
Users ⁶	8,903	8,723	95,820	90,851	74,958
Random user agent on browser start	✓	✓		✓	
Random user agent by time interval	✓	✓		✓	
Spoofs user agent in HTTP headers	✓	✓	✓	✓	✓
Spoofs user agent in navigator	✓	✓	✓	✓	✓
Spoofs additional <code>navigator.*</code> properties	✓ ⁷			✓	✓

(UAS = User-Agent Switcher)

¹ <https://addons.mozilla.org/en-US/firefox/addon/chameleon-ext/>

² https://addons.mozilla.org/en-US/firefox/addon/random_user_agent/

³ <https://addons.mozilla.org/en-US/firefox/addon/user-agent-switcher-revived/>

⁴ <https://addons.mozilla.org/en-US/firefox/addon/uaswitcher/>

⁵ <https://addons.mozilla.org/en-US/firefox/addon/user-agent-string-switcher/>

⁶ According to Mozilla's extension archive (last visited on 03/26/2020).

⁷ Only with the setting that activates script injection.

(see also [Appendix C](#)) into each visited page. This allows more advanced spoofing, e.g. for additional `navigator` objects besides `userAgent`.

5.2.2. *fingerprint2.js* analysis

To analyze how successful each extension spoofs the chosen user agent, we use the commonly used open source tool *fingerprint2.js* to measure each extension while it is added to OpenWPM. For simplicity, the output of *fingerprint2.js* attributes is truncated at the 501st character. Further, besides *Chameleon* and its script injection mode, all tools are tested in standard settings. The only modified setting is the user agent to spoof. We choose Chrome on Windows as our desired user agent because the test machine runs on Ubuntu with OpenWPM (and thus Firefox). A different platform and browser family reveals most differences.

Results. The interesting *fingerprint2.js* test cases considering inconsistent spoofed values are depicted in Table 5.3.

Table 5.3.: *fingerprint2.js* analysis of the five popular user agent spoofing extensions from the Firefox extension archive.

<i>fingerprint2.js</i> integrity check for...	OpenWPM (desired)	Firefox spoofing extension				
		Chameleon	Random User-Agent	UAS by Linder	UAS by Alexander Schlarb	UAS and Manager
Languages	–	–	–	–	–	–
Resolution	–	–	–	–	–	–
Operating system	–	¹	faked	faked	–	–
Browser	–	faked	faked	faked	faked	faked

– : This fingerprint information is not suspicious.

faked: *fingerprint2.js* detected this fingerprint information is faked respectively not consistent with the full browser fingerprint.

(UAS = User-Agent Switcher)

¹ *Chameleon* is only detected to fake the operating system if it runs without optionally activated script injection.

None of the tested extensions triggers *fingerprint2.js*'s language or resolution integrity test. This is expected as none of them changes the browser language (at least in standard settings) or spoofs another resolution (also, at least not in standard settings).

However, three extensions are detected to lie about the operating system. *Chameleon* suffers from this issue only with deactivated script injection. But *Random User-Agent* and *User-Agent Switcher by Linder* are also detected. Considering their functionality, this is explainable because they both do not spoof additional `navigator` properties such as `platform`. Thereby, the inconsistency between the user agent and `platform` is detected by *fingerprint2.js*.

The last check for the browser integrity classifies all extensions as “faked”. How can this happen? The problem lies in different implementations of the standard JavaScript function `toString` across Firefox and Chrome. Therefore, a spoofed Chrome in an actual Firefox can be revealed and vice versa. It may not be impossible but at least difficult to spoof this correctly because the whole `toString` function would need to be spoofed. Further details about this are mentioned in [Section 5.3](#).

5.2.3. Conclusion

To get to the point, the more sophisticated extensions *Chameleon*, *User-Agent Switcher* by Alexander Schlarb and *User-Agent Switcher and Manager* perform best. Interestingly, this does not necessarily go along with their number of users. Anyway, the extensions show how it is possible to spoof the user agent, and they thereby can reduce OpenWPM’s detectability. Nevertheless, they represent no single solution because we also need to consider cross-site tracking. Therefore, they need to spoof different identities not only based on random intervals but using a more sophisticated method like FP-Block does.

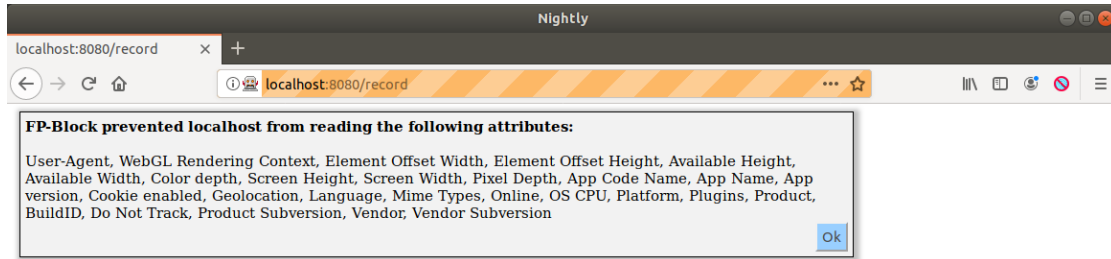
5.3. State-of-the-art countermeasure: FP-Block 2.0

The academic state-of-the-art tool FP-Block 2.0 [TJM15; CY19] for countering fingerprint-based re-identification is tested in how far it works in combination with OpenWPM. Ideally, the extension helps to counter OpenWPM’s detection, specifically regarding cross-site tracking.

General remarks. During our experiments with FP-Block 2.0, we noticed two interesting things. First, by using FP-Block 2.0, queries for DOM attributes took much longer. Instead of a few seconds, JavaScript template attacks, despite deactivated OpenWPM instrumentation, took around two minutes. This enables timing attacks. However, the focus of this thesis lies on property based detection.

Secondly, FP-Block 2.0 informs the user about the attributes it spoofs and blocks. This happens by use of a message box that is injected to the page DOM, see [Figure 5.4](#). The problem is that this message can be detected by websites because it is part of the page DOM. A solution is to disable “Show notifications” in the visual plugin settings or, even better for automation, modify the `DEFAULT_OPTIONS` in the extensions source file `src/utils/constants.js` to disable the option even before installing the extension.

Benefits. FP-Block’s big advantage is the creation of a huge number of different fingerprint profiles that are all coherent but still not the same. In doing so, FP-Block creates authentic fingerprint profiles because it considers many attributes to generate



JavaScript Template Attack

Running, please wait...

Figure 5.4.: Message FP-Block 2.0 shows and injects into the page DOM when it blocks a page from reading attributes.

the profiles. That functionality compared with the clever approach to separate fingerprint profiles based on the visited domain makes it very useful to counter cross-site tracking.

Integration problems. The current FP-Block 2.0 implementation is not suitably designed to work together with OpenWPM. In OpenWPM, each website is visited by a new and reset browser instance. Data is not preserved against browser instances. Therefore, FP-Block would newly generate the list of fingerprints each time another website is visited. Furthermore, FP-Block’s assignment of fingerprint profiles to domains is also not preserved. Pages visited twice would not be visited with the same fingerprint profile again.

***fingerprint2.js* analysis.** *fingerprint2.js* is used to check FP-Block’s fingerprint integrity. For some fingerprint profiles chosen by FP-Block 2.0, *fingerprint2.js* recognize that the browser indicated by the user agent is spoofed. Looking into *fingerprint2.js*’s code shows that the problem is the `navigator.productSub` property. With FP-Block 2.0, the property is **undefined**, while for Chrome based browsers (e.g. Google Chrome or Opera) it equals `20030107` and for Gecko based browsers (e.g. Firefox) it equals `20100101`⁶.

This is a problem of FP-Block 2.0, unrelated to OpenWPM. Fixing it is no easy task, as *fingerprint2.js* also checks whether the product subversion is coherent with the `toString` function behavior of the corresponding browser family. Previous work Nikiforakis et al. [Nik+13] and Vastel et al. [Vas+18] already identified this detection method that Gecko based browsers add two newline characters to the string representation of functions, while Chrome based browsers do not add newline characters. A potential solution would be to

⁶<https://html.spec.whatwg.org/multipage/system-state.html#dom-navigator-productsub> (last visited on 03/23/2020)

overwrite `Function.prototype.toString` with a JavaScript proxy that resembles the printing style of the chosen fingerprint. Alternatively, for OpenWPM one could configure FP-Block 2.0 such that it only generates fingerprint profiles with Firefox browsers. A downside is this would shrink the available number of profiles drastically.

Spoofing a completely coherent set of attributes in general is a complex task because of the deviating behavior and APIs across different browsers. Mulazzani et al. [Mul+13] could already identify different JavaScript engines using Test262⁷ as JavaScript engines behave differently to this test.

Fingerprint surface based analysis. The fingerprint surface based measurement framework, without changing FP-Block’s standard settings, reveals differences between OpenWPM headful and OpenWPM headful with FP-Block 2.0 (both with deactivated OpenWPM instrumentation).

Several `window` keys originating from FP-Block 2.0 are added:

- `key`
- `isHeightDetected`
- `isWidthDetected`
- `original_fillText`
- `original_strokeText`
- `original_toDataURL`
- `thisScriptElement`
- `detected`
- `generateNoise`
- `WebGLRenderingContext`

Some of them, such as `WebGLRenderingContext`, are spoofed on purpose. In this example, FP-Block 2.0 removes the ability to access WebGL and the information about the system WebGL would otherwise reveal. Other keys such as `thisScriptElement` or `generateNoise` are objects created for FP-Block’s functionality. They raise OpenWPM’s detection surface.

Also, the order of properties of the `window.navigator` object is not preserved. Values touched by FP-Block 2.0 are listed earlier. In [Chapter 6](#), we solve similar spoofing of OpenWPM’s `webdriver` attribute by using a JavaScript proxy for overwriting.

⁷<https://github.com/tc39/test262> (last visited on 03/23/2020)

JavaScript template attack. Without changing the standard settings of FP-Block 2.0, we perform a JavaScript template attack to compare OpenWPM headful and OpenWPM headful with FP-Block 2.0 (both with deactivated OpenWPM instrumentation). In total, 3468 properties differ in a direct comparison. However, it is FP-Block’s functionality to spoof and block certain values. Therefore, several of the 3468 differences are desired.

But there are also differences that make OpenWPM more suspicious. For example, the string representation of the `window.canvas.toDataURL` function changes from

```
1 function toDataURL() { [native code] }

to

1 function() {
2   generateNoise(this);
3   var result = original_toDataURL.apply(this, arguments);
4   detected('localhost', 'Canvas To Data URL', 'toDataURL',
   ↪ 'block', result);
5   return (result);
6 }
```

Several other functions are changed the same way.

Further, FP-Block 2.0 introduces several new DOM elements such as `thisScriptElement` and its sub-properties that cause detectable deviations. However, they were already discussed in the previous paragraph [Fingerprint surface based analysis](#).

Conclusions. Summarizing, it is justified that FP-Block 2.0 is a state-of-the-art anti cross-site tracking tool because of the extensive fingerprint profiles and their separation based on visited domains. Unfortunately, a unmodified FP-Block cannot be reasonably integrated to OpenWPM because each browser instance of OpenWPM is reset. It is also detectable on the JavaScript level because it adds new objects to the page’s DOM and changes existing ones.

We propose to modify FP-Block 2.0 such that it becomes usable in combination with OpenWPM and to change its JavaScript spoofing methodology to become more sophisticated. [Chapter 6](#) finds the best JavaScript spoofing method for this purpose.

6. Detectability of different JavaScript spoofing methods

As shown in [Chapter 5](#), existing countermeasures for reducing OpenWPM’s detectability are insufficient. In this chapter, we investigate how to spoof attributes on the level of JavaScript without websites being able to recognize the attributes are tampered with.

OpenWPM contains the `navigator.webdriver` attribute that is widely used for web bot detection [[JKV19](#)]. As encountered in [Chapter 4](#), it is also a stable attribute. Therefore, that attribute is investigated.

6.1. Methodology

Solving stealthy spoofing on the level of JavaScript has two advantages:

1. It allows spoofing of various JavaScript properties and is not related to a specific one. Such a general method of spoofing can be reused in other projects such as FP-Block to become more advanced against detectors of JavaScript tampering.
2. Specifically for OpenWPM, a JavaScript solution is much easier to maintain than other possible measures such as source code modification and re-compilation of Firefox.

To address stealthy JavaScript spoofing, this chapter investigates different methods to spoof OpenWPM’s `webdriver` property. In this context, a *spoofing method* is a variant how to overwrite an already existing JavaScript property, such as `webdriver`, with another value. Therefore, the new option `hide_webdriver` is added to OpenWPM’s Firefox instrumentation extension that, if enabled, additionally loads our JavaScript content script¹ into each visited web page. The content script contains the respective JavaScript code responsible for spoofing.

In order to compare the stealthiness of the approaches, their modifications are measured using both the fingerprint surface based measurement framework by [[JKV19](#)] (*BrowserBasedBotFP*) and JavaScript template attacks [[SLG19](#)]. Both measurement methods can compare two captured browser environments against each other. For our objective, a captured baseline environment – that of a regular Firefox instance – is compared to OpenWPM and the different integrated spoofing methods.

¹The concept of page- and content-scripts for Firefox extensions is described in [Appendix C](#).

Each of the two measurement methods covers a separate aspect:

- *BrowserBasedBotFP* captures differences of fingerprinting attributes already known to the literature, such as `window` keys or fields of the HTTP header.
- JavaScript template attacks find all static differences in the JavaScript environment. Thus, they not only cover properties one already suspects to be different, but they allow for a more systematic approach. However, template attacks are neither complete. They cannot recognize differences that need an explicit previous action, such as needed for audio or canvas fingerprinting. In terms of Dolfing’s [Dol19] taxonomy, behavior based fingerprinting methods are not captured by template attacks.

This detailed and strict measurement criteria allow to be ahead in the arms race between spoofing and anti-spoofing detection. At best, the found solution would render anti-spoofing detection impossible at all, and not only set it one or two steps ahead.

6.2. About the `webdriver` attribute

In OpenWPM, the `navigator.webdriver` JavaScript property is set. It is defined inside the W3C’s WebDriver working draft. It states the `navigator.webdriver` property initially equals `false` and is set to `true` if “the user agent is under remote control”.² In other words, it is activated if a browser is automated.

As OpenWPM uses Selenium WebDriver to automate Firefox, this option is set. In comparison, the `webdriver` attribute equals `false` for a regular Firefox instance.

In contrast to Firefox, Chrome handles the `webdriver` property in another way. There, it is only defined in the `true` case and otherwise the DOM does not contain the property at all (without browser automation JavaScript code would thus return `undefined`). Note this behavior of Chrome does not follow the W3C working draft.

Other options to address the `webdriver` attribute for OpenWPM. Although this chapter discusses how to spoof using a Firefox extension and JavaScript, we want to sketch alternative approaches to modify OpenWPM’s `webdriver` property.

One could think a network proxy can forward all traffic but change traffic containing the `webdriver` property. However, the property originates from the JavaScript level executed inside the client’s browser. Hence, website’s JavaScript code can obfuscate or even encrypt a message containing the read out value such that a network proxy cannot detect or inspect the attribute anymore. Therefore, network proxies are no suitable solution.

²<https://www.w3.org/TR/2019/WD-webdriver2-20191124/#dom-navigatorautomationinformation-webdriver> (last visited on 01/28/2020)

A possible approach is to change and compile Firefox’s source code. Vlot [Vlo18] did something similar by re-compiling the ChromeDriver (WebDriver implementation for Chrome) to alter Chrome’s hard coded `$cdc_asdjflasutopfhvcZLmcfl_` string that identifies ChromeDriver driven browsers. For Firefox, only one line of code³ would need to be adjusted such that `false` is returned in all cases:

```
1 bool Navigator::Webdriver() {
2     return Preferences::GetBool("marionette.enabled", false); //Here
3 }
```

Unfortunately, OpenWPM uses Firefox unbranded builds (as described in Section 2.4). We tried to compile Firefox unbranded ourselves but this is not straightforward, the compilation would need to be maintained in OpenWPM and adds a huge overhead to the initial setup of OpenWPM. Still, compilation remains as last choice that could address OpenWPM’s problem with the `webdriver` attribute.

6.3. Spoofing methods

To find suitable JavaScript spoofing methods, we performed a literature review and, additionally, investigated how the five popular user agent spoofing extensions already encountered in Section 5.2 and FP-Block implement spoofing.

The identified options to spoof the `webdriver` property are:

1. Use `Object.defineProperty` to overwrite the property. Storey et al. [Sto+17] for example use this method to overwrite JavaScript’s universal `toString` function to hide DOM modifications made by their ad blocker.
2. Use `Object.prototype.__defineGetter__` to overwrite the property. This is the predecessor of `Object.defineProperty`.
3. Set a JavaScript prototype for the `webdriver` property to overshadow the real value.
4. Replace `window.navigator` with a JavaScript proxy that forwards all queries besides that for `webdriver` to the original `navigator` object. JavaScript proxies are designed to change the behavior of an existing object, while it remains impossible to see the difference between the original and proxied version of the object⁴.

The corresponding code and technical details are discussed in Appendix F.

³<https://searchfox.org/mozilla-central/rev/5e830ac8f56fe191cb58a264e01cdbf6b6e847bd/dom/base/Navigator.cpp#1973-1975> (last visited on 01/28/2020)

⁴https://exploringjs.com/es6/ch_proxies.html#sec_detect-proxies (last visited on 01/30/2020)

6.4. Comparison of spoofing methods

Table 6.1 compares the results of the four spoofing methods. At best, no feature would be detected for a spoofing method.

Table 6.1.: Comparison of the detectability of different JS spoofing methods.

Spoofing method	Detected feature				
	Incorrect order of navigator properties	Modified navigator._length	New <code>Object.keys(navigator)</code>	Defined navigator.__proto__	Unnamed <code>window.navigator</code> functions ¹
Desired (regular Firefox)	–	–	–	–	–
<code>Object.defineProperty</code>	✓ ²	✓	? ³	–	–
<code>Object.prototype.__defineGetter__</code>	✓	✓	✓	–	–
<code>Object.setPrototypeOf</code>	–	–	–	✓	–
JavaScript proxy	–	–	–	–	✓

✓: Feature detected

–: Feature not detected

¹ The JavaScript proxy method causes 20 native `window.navigator` functions to become anonymous functions, that is they lose their function names but functionality is preserved.

² For `enumerable = true`, the order of the `webdriver` property is altered. Otherwise, for `enumerable = false`, it is not enumerated at all. This is also considered to be of wrong order.

³ For `enumerable = false`, `Object.keys(navigator)` is empty. Otherwise, it does contain the additional `webdriver` key.

`Object.defineProperty` method. The `defineProperty` method increments the `_length` property of `navigator` by 1. Experiments comparing regular Firefox with spoofed OpenWPM show that for each property overwritten using `Object.defineProperty()`, `window.navigator._length` is incremented by 1 – irrespective of the number of times each unique property was overwritten. So, if one overwrites `userAgent` 10 times and

`webdriver` 2 times, the `_length` property will only be incremented by 2.

Further, while it is possible to rewrite this `_length` property, e.g. by using `defineProperty`, JavaScript template attacks can still reveal the value of the original `_length` property because they traverse the DOM's prototype chain.

A further problem of `defineProperty` is that it distorts the enumeration order of the `navigator` object by listing modified properties first, as illustrated in Figure 6.2.

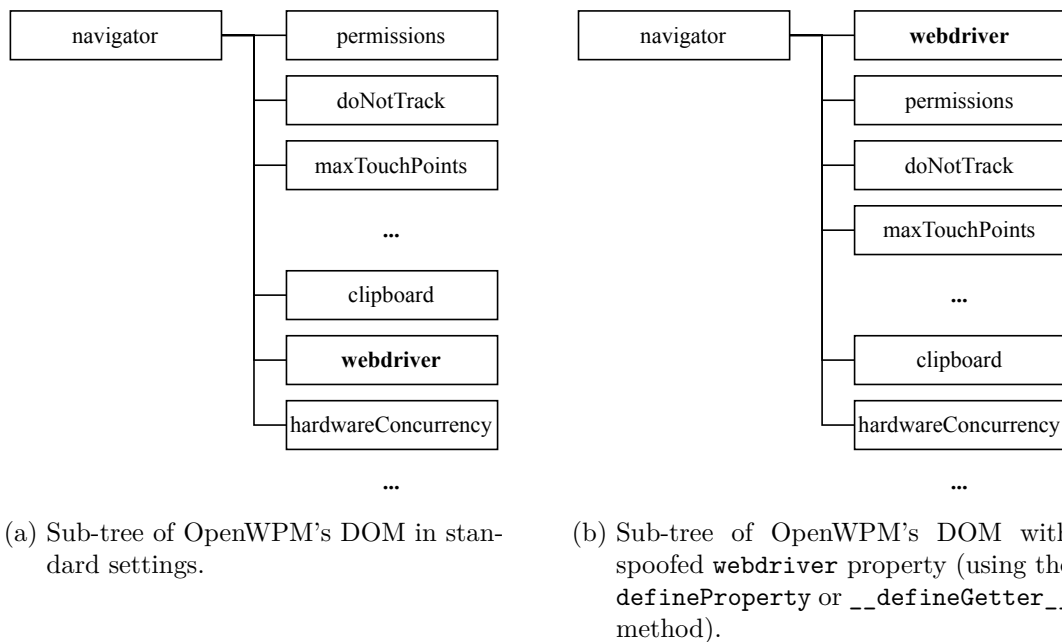


Figure 6.2.: Comparison of `navigator` sub-trees of OpenWPM's DOM, with and without spoofing of the `webdriver` attribute.

While research shows the property order in JavaScript is not reliable in each case⁵, in our tests the modified attribute is always listed first. It could thus be used to detect the modification. Vastel et al. [Vas+18] for example used the `navigator` order to detect different browsers. Unfortunately, it is impossible to directly change the order of JavaScript properties. But it is possible to redefine all other `navigator` properties too, without changing their actual value. That way the original order could be restored. However, even then this method still remains detectable as it causes the `navigator._length` property to be incremented by 1.

⁵<https://stackoverflow.com/questions/5525795/does-javascript-guarantee-object-property-order/> (last visited on 01/30/2020)

Moreover, `defineProperty` alters whether `webdriver` is an enumerable JavaScript property. Without spoofing, an enumeration of `navigator` contains the `webdriver` property but it is not listed in `Object.keys(navigator)`.

On the one side, with `defineProperty`'s option `enumerable = true`, the spoofed `webdriver` is still regularly enumerated. However, `Object.keys(navigator)` then contains the key `webdriver`.

On the other side, `enumerable = false` prevents the new key in `Object.keys(navigator)`. However, `webdriver` then is not enumerated anymore for `navigator`. Interestingly, in that case `window.navigator._length` is still incremented. Therefore, it cannot be that an incremented `_length` corresponds one-to-one with the keys contained in `Object.keys(navigator)`.

Anyway, both `enumerable = true` and `enumerable = false` cause detection.

`Object.prototype.__defineGetter__` method. This method is detectable by the same three characteristics as the `defineProperty` method. Only the option `enumerable` is not applicable. `__defineGetter__` always adds a new key to `Object.keys(navigator)`.

`Object.setPrototypeOf` method. Compared to `defineProperty` and `__defineGetter__`, the prototype overwrite does not cause their previous three issues. However, it necessarily sets `window.navigator.__proto__.webdriver` because that is how prototype overwrites work. In comparison, for regular Firefox, `__proto__` is not defined on `navigator.webdriver` at all.

As `window.navigator.__proto__.webdriver` can be directly accessed and it is not necessary to do so via the prototype chain (e.g. using template attacks), this form of spoofing the `navigator.webdriver` attribute is easy to detect.

JavaScript proxy method. Finally, the JavaScript proxy transform 20 native `navigator` functions into anonymous functions. In other words, they lose their function name whereas all functionality stays the same. The difference is detectable by using the `toString` function on the affected functions such as `vibrate`:

```
1 function vibrate() { [native code] } // In regular Firefox
2
3 function () { [native code] } // Same function, but in
  ↪ OpenWPM with navigator being a proxy
```

This happens as we explicitly need to clone a set of functions (such as `vibrate`) into the proxy's scope because they otherwise brake functionality of more complex web pages that for example embed videos. For instance, advertisements and videos otherwise did not work on <https://www.reddit.com/r/videos/>. They caused several errors such as the following:

```
1 TypeError: 'javaEnabled' called on an object that does not implement  
  ↪ interface Navigator.
```

We found no way to clone these functions correctly to the page scope without them losing their name.

6.5. Conclusions

The two most promising solutions are the prototype overwrite and the JavaScript proxy because they are least detectable.

On the one side, using the prototype overwrite has the advantage that it certainly does not break functionality and it is less complex. However, it is easy to detect as the identifying `window.navigator.__proto__.webdriver` property is set.

On the other side, the proxy variant is detectable by JavaScript template attacks but therefore more powerful as it also allows implementing more complex spoofing logic.

Unfortunately, combining two or more of the presented spoofing methods is neither a suitable solution as this only adds up the detectable features while not removing any of them.

7. Proof of concept implementation

In this chapter, a prototype of the JavaScript proxy method derived in [Chapter 6](#) is implemented in OpenWPM to spoof its `webdriver` property. We additionally extend the prototype with Alexander Schlarb's countermeasure against un-spoofed attribute leakage, as will be described below. The combined prototype version led to a pending contribution (pull request) for the OpenWPM framework.¹

The proof of concept implementation serves two purposes. On the one side, it shows that the prototype works outside our used measurement tools. On the other side, it points out whether spoofing OpenWPM's `webdriver` property is relevant in practice.

7.1. The prototype

The main component of the prototype is a JavaScript proxy that replaces the DOM's original `navigator` object. It is the most stealth JavaScript spoofing method investigated in [Chapter 6](#). Its core functionality is given by the code below:

```
1  if (prop === "webdriver") {
2      return false;
3  } else {
4      let value = Reflect.get(origNavigator, prop); // Get value that
5      ↪ would be returned by the regular navigator object
6      if(typeof(value) === "function") {
7          // Explicitly clone functions to make them work in the proxy
8      }
9      return value;
10 }
```

Whenever `navigator`'s `webdriver` property is queried, a spoofed value of `false` is returned (OpenWPM's standard `webdriver` value is `true`, as described in [Section 6.2](#)). Otherwise, the query is forwarded to the original `navigator` object. The omitted code in line 6 is necessary to explicitly bring functions of the original `navigator` object into the proxy's scope. Otherwise, the function calls may fail, as described in [Section 6.4](#).

¹<https://github.com/mozilla/OpenWPM/pull/526/> (last visited on 03/23/2020)

7. Proof of concept implementation

The detailed code can be found in [Appendix F](#). Besides glue code necessary for the proxy to work in a Firefox extension, it additionally contains a countermeasure against access to the un-spoofed `webdriver` property if it is accessed in iframes.

The problem with iframes. During study of the five popular user agent spoofing extensions from [Section 5.2](#), we came across an interesting discussion in the forum of *User-Agent Switcher by Alexander Schlarb*². They pointed out that values are not spoofed inside iframes. The only reference to this problem we identified in the literature is the work of Raschke and Küpper [[RK18](#)]. Raschke and Küpper analyzed canvas fingerprinting and therefore needed to overwrite JavaScript APIs. They copied functionality from the Firefox and Chrome extension *Canvas Defender* to overwrite the APIs such that they are also overwritten in iframes. However, *Canvas Defender*'s development seems to be suspended.³

In [Appendix E](#), we conduct an experiment that uses Alexander Schlarb's tests to see whether the other four popular user agent spoofing extensions contain measures against the leakage of un-spoofed properties inside iframes. The experiment shows the countermeasures by *User-Agent Switcher by Alexander Schlarb* work fairly well. Therefore, as a first step, we added his countermeasure against the iframe leak to our spoofing method. But further research needs to be done how to best counter the problem of iframes. We found discussions that show some users were already aware of this problem back in 2015⁴. Also, other extensions such as *CanvasBlocker by kkapsner*⁵ implemented specific countermeasures, too⁶.

The problem with `window.open('')`. In the same forum discussion, a second problem attracted our attention. Permitted browser pop-ups enable access to the original property value. How does this work?

JavaScript allows to open a new window. The reference to the new window can be used to access its attributes. For example, the JavaScript command `window.open('').navigator.webdriver` returns the `webdriver` value out of the new window's scope.

Unfortunately, the property is not spoofed in this new window and the original value is revealed. We could not identify why this happens. A new `about:blank` page is

²<https://gitlab.com/ntninja/user-agent-switcher/-/issues/9> (last visited on 03/23/2020)

³The extension was last updated in 2017. For Firefox, at the 9th and for Chrome at the 11th of July (see <https://addons.mozilla.org/en-US/firefox/addon/no-canvas-fingerprinting/> and <https://chrome.google.com/webstore/detail/canvas-defender/obdbgnebcljmgkoljcdddaopadkifnpm> (last visited on 04/02/2020)).

⁴<https://github.com/dillbyrne/random-agent-spoofers/issues/309> (last visited on 04/02/2020)

⁵<https://addons.mozilla.org/de/firefox/addon/canvasblocker/> (last visited on 04/02/2020)

⁶<https://github.com/kkapsner/CanvasBlocker/blob/master/lib/iframeProtection.js> (last visited on 04/02/2020)

opened and injection should happen to it, too. But even in the browser console of the opened page, the `webdriver` property is not spoofed. In comparison, the property is well spoofed inside a manually opened new window (via Ctrl+N or the menu) that also opens `about:blank`.

In regular Firefox, the default setting prevents pop-ups, so this method does not work. However, OpenWPM deviates from this setting and allows pop-ups by default⁷.

Our stealthy prototype of OpenWPM addresses this issue and sets the standard setting to disallow pop-ups⁸, which resolves this issue for OpenWPM.

7.2. Validation experiment

First of all, the prototype is only tested for OpenWPM in headful mode. We suspect the closer to a headful browser OpenWPM becomes, the more the performance also resembles that of a headful browser. Hence, it is not affordable to focus on improving OpenWPM in headless mode because this in the end will require re-implementing headful browser functionality. In [Section 9.2](#), we describe a way it is nevertheless possible to run our improved headful version under headless circumstances, combining both advantages.

Ideally, the stealth prototype would be validated automatically. Our concept included using Vlot’s [\[Vlo18\]](#) scanner to identify all Tranco top 10,000 pages [\[Le +19\]](#) that access the `webdriver` property. That pages should be visited automatically by regular and stealth OpenWPM. Differences would be encountered by comparing the screenshot of both crawls. However, especially the automation turned out to be the bottleneck. The screenshots could not be compared in a satisfying quality and did not capture differences that occur during manual surfing.

Hence, we manually address the validation in the first place. As starting point, we have to find websites that change their behavior for an activated `webdriver` property. Very few could be derived by analyzing the screenshots of the “failed” automated experiment. To find additional ones, we conducted internet discussions dealing with `webdriver` spoofing. There, the authors often list example pages where they encountered problems with being detected.

⁷This seems to originate from Selenium, see https://github.com/SeleniumHQ/selenium/blob/8a0691f55df5868b518e0f53c3c90b6251607dd4/third_party/js/selenium/webdriver.json#L28 (last visited on 01/30/2020).

⁸See <https://github.com/FInch/OpenWPM/commit/4ca525ca0200765f3d4a70a23d15516c2498344e> (last visited on 01/30/2020)

7. Proof of concept implementation

The pages suspected to trigger on the active `webdriver` property are manually visited with:

1. Normal Firefox;
2. Regular OpenWPM in headful mode without its instrumentation;
3. Our spoofing prototype in headful mode without OpenWPM's instrumentation.

First, we manually surf to the pages using normal Firefox to observe their ordinary behavior. Second, the pages are manually observed with OpenWPM without our spoofing modification to show that they trigger web bot detection in some way, for example by showing a CAPTCHA. Third, we use our OpenWPM prototype to manually visit the same pages to show that the web bot detection does not occur if the `webdriver` property is spoofed.

All three stages of the experiment are documented by screenshots. Between them, a cool-down time of several hours is added to exclude other types of fingerprinting as cause for different observed behavior. This helps as websites exclude detected bots and other types of website visitors often only temporarily. Additionally, each stage is performed from an internet connection with a new IP address.

Finally, we tested that our prototype does not interfere with common websites where it should cause no difference. Therefore, we manually inspected ordinary websites by surfing on them with our prototype. We encountered no unintended side effects.

7.3. Results

In the following, we present pages that were found to trigger web bot detection with a present `webdriver` property but that behave ordinary if visited with a spoofed `webdriver` property. Pages suspected in doing so but where we found no evidence are omitted.

[Table 7.1](#) shows the different types of deviating behavior we encountered that our prototype can prevent for the found pages by spoofing the `webdriver` property. They are introduced with examples for each category. Further screenshots can be found in [Appendix G](#).

Site access blocked. This category contains pages that block accessing them if `webdriver` is present. Some pages detect this instantly on their start page, while others only start to block after a few sub pages are visited. [Figure 7.2](#) shows an example of the type of messages shown in such cases.

Site access blocked (with CAPTCHA for un-block). Similarly to the previous category, access to the page is blocked. However, this pages allow the visitor to proof he is human by completing a CAPTCHA. An example is given in [Figure 7.3](#). Regarding

Table 7.1.: Types of deviating behavior that regular OpenWPM encounters and our prototype prevents.

	<i># sites</i>	
	<i>Regular OpenWPM</i>	<i>Stealth OpenWPM</i>
Pages visited in total	14	14
<i>Type of deviating behavior</i>		
Site access blocked	6	0
Site access blocked (with CAPTCHA for un-block)	3	0
Login only with CAPTCHA	3	0
Other deviating behavior	2	0

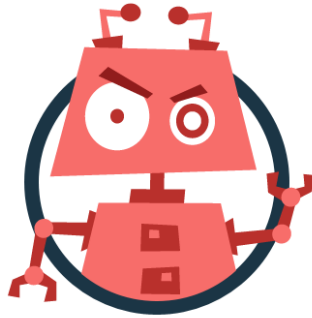
CAPTCHAs, Sivakorn et al. [SPK16] showed that Google reCAPTCHA considered user agents of outdated browsers to be suspicious. In such cases, the system requested the visitor to solve a CAPTCHA, while otherwise forwarding him without solving one.

Login only with CAPTCHA. These pages show CAPTCHAs on their login pages. While this is not unusual and some websites always require this to prevent spam, the tested pages of this category only do so if the `webdriver` property is present. All pages use reCAPTCHA by Google that seems to trigger on this property. Figure 7.4 depicts an example.

Other deviating behavior. Here, we encountered behavior not fitting into the previous three categories. `stubhub.com` does not block anything, but the site loads its content delayed by 10 to 30 seconds. Regular surfing is almost impossible as you need to wait this time after each click. On the same network but with a different machine and for our prototype, this behavior did not occur.

The other page is `google.com`. For normal Firefox and our prototype, logging into a Google account works without problems. For regular OpenWPM however, one is not permitted to log in with the message shown in Figure 7.5.

7. Proof of concept implementation



Unable To Identify Your Browser

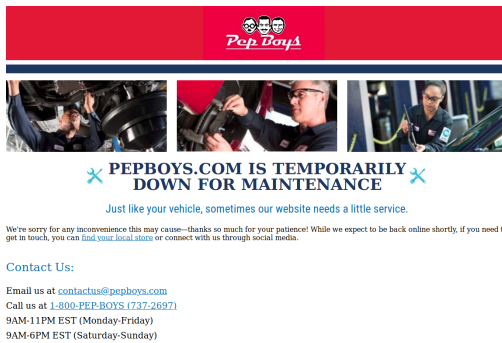
As you were browsing <https://www.controller.com> something prevented us from identifying your browser. There are a few reasons this might happen:

- You've disabled JavaScript or cookies in your web browser.
- A third-party browser plugin, such as Ghostery or NoScript, is preventing JavaScript from running. Additional information is available in this [support article](#).
- A browser plugin, such as Google Chrome's Data Saver, is routing your network traffic through a data center or other proxy. Additional information is available in this [support article](#).

Resolving these issues should restore your access to www.controller.com.

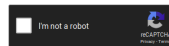
You reached this page when attempting to access <https://www.controller.com/listings/aircraft-for-sale-in-hawaii/?categoryId=13&country=usa&eventtype=for-sale&state=hawaii> from 93.237.239.240 on 2020-03-25 11:57:07 GMT.
Trace: ftdb4066-b19b-4154-b8ce-ccbea8710fea (409) via 449bb29d-9aa5-44ea-a964-418570a62186; 0.002 / 2275

Figure 7.2.: Blockage on [controller.com](https://www.controller.com).



(a) First, it is stated the page is “temporarily down for maintenance”.

Please verify you are a human



Access to this page has been denied because we believe you are using automation tools to browse the website.

This may happen as a result of the following:

- Javascript is disabled or blocked by an extension (ad blockers for example)
- Your browser does not support cookies

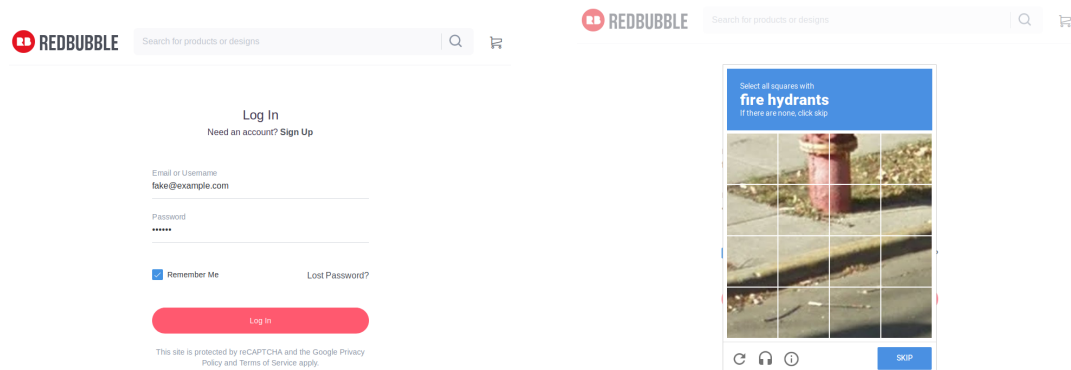
Please make sure that javascript and cookies are enabled on your browser and that you are not blocking them from loading.

Reference ID: #bbbf3d0-6e97-11ea-8f62-e1301c9b3b5a



(b) After refreshing the page, a block message with CAPTCHA is shown.

Figure 7.3.: Blockage with offered CAPTCHA on [pepboys.com](https://www.pepboys.com).



(a) Entered credentials for login.

(b) Before the wrong credentials are checked for correctness, one needs to solve a CAPTCHA.

Figure 7.4.: CAPTCHA for login on [redbubble.com](https://www.redbubble.com).

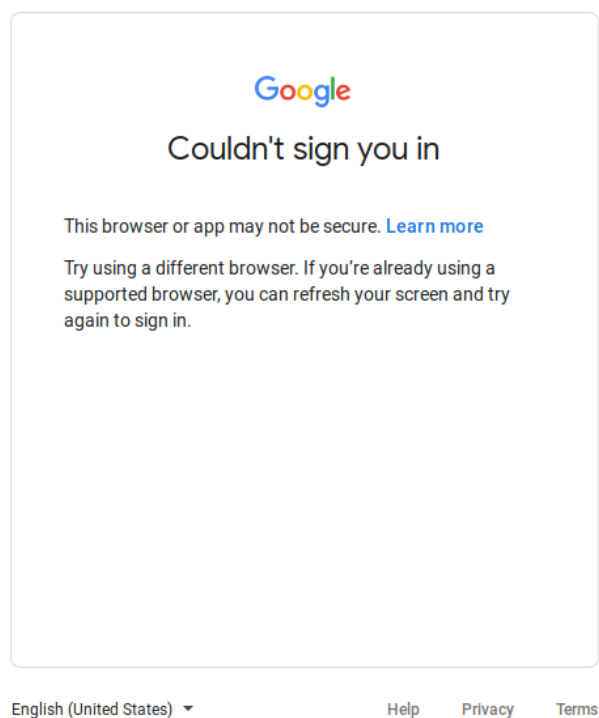


Figure 7.5.: Prevented login on [google.com](https://www.google.com).

8. Future work

OpenWPM framework. Our work shows OpenWPM can be revealed by the deviated position of the `referer` attribute inside HTTP request. Future research needs to investigate whether this detection feature remains stable over time. If this is the case, possible solutions to restore the original attribute order include a network proxy or Firefox extension modifying the headers.

With the current stealth version of OpenWPM, it is still possible to access the unspoofed value of the `webdriver` property if accessed by a specifically crafted dynamic iframe (see [Section 7.1](#)). As some other spoofing or blocking extensions also deal with that problem, it may be possible to find a more advanced method to be completely robust against value disclosure via iframes.

Further, this thesis focused on property based web bot detection. However, behavioral detection analyzing OpenWPM's actions can similarly disclose the browser to be a web bot, for example by the speed information is entered to input fields. There is potential to further improve OpenWPM's stealthiness in this area.

Studies using OpenWPM. The most covert form to run stealth OpenWPM with `webdriver` spoofing is in headful mode with an explicitly adjusted window size. Previous studies using the OpenWPM framework can be repeated in this setting to measure the extent web bot detection influenced the results of the original studies. Moreover, future research can benefit from the stealth version to reduce the risk that web bot detection influences its results.

Improving FP-Block. Current FP-Block 2.0 can be improved to become more advanced.

On the one side, FP-Block does not spoof the `navigator.productSub` property but blocks it. Therefore, *fingerprint2.js*'s integrity check triggers and reports a spoofed browser. This can easily be fixed by spoofing the correct value for `productSub`, which actually is a constant for each different browser family.

However, *fingerprint2.js* then still might report a spoofed browser as Firefox and Chrome based browsers behave differently when printing a function by calling its `toString`. It needs to be researched for the best approach to change the behavior of the `toString` function accordingly to the spoofed browser family. Overwriting it is possible but it is difficult to do so without being able to detect the alteration.

On the other side, FP-Block's way of spoofing DOM properties can be improved. Although it is not the main objective of FP-Block to be robust against every anti-spoofing detector, using JavaScript proxies can already reduce detection via template attacks. It needs to be checked whether this also helps to hide the DOM objects introduced by FP-Block against template attacks. Further, we did not find a complete solution for the access of un-spoofed attributes via (dynamic) iframes. This still needs to be investigated in more depth.

9. Conclusions

9.1. Results and answers to research questions

Analyzing the stability of OpenWPM’s fingerprint surface showed the main identifying properties that allow detecting OpenWPM, such as the `webdriver` property, are stable across major OpenWPM versions. We also confirm the findings of Krumnow et al. [Kru+] that both the headless mode and OpenWPM’s JavaScript instrumentation introduce most detectability. With the position of the `referer` HTTP header field, a new but yet to be considered as unstable detection method was found.

Existing measures against detection turned out to be able to improve OpenWPM’s detectability but no exclusive solution was capable of solving the problem entirely. We also revealed possible flaws in existing user agent spoofing extensions and FP-Block [TJM15; CY19], considering their spoofing methods and the possibility to access un-spoofed values via (dynamically created) iframes.

While we focused on spoofing at the level of JavaScript to remove detection by the `webdriver` property, we found JavaScript proxies being the most advanced and stealthy solution. Unfortunately, a small detection surface remains even for this method.

Finally, we implemented a stealthy version of OpenWPM that addresses the `webdriver` property by overriding the DOM’s `navigator` object with a JavaScript proxy. Manually validating the stealthy version on actual websites shows it successfully spoofs the property. Further, the `webdriver` property turned out to be the determining factor for several manually discovered pages whether they classify a visitor as web bot or not. Concluding, our new stealthy version is certainly less distinguishable from a regular web browser.

9.2. Impact of results

OpenWPM. Iframes can reveal the un-spoofed value of JavaScript properties (see [Section 7.1](#)). This also applies to OpenWPM’s instrumentation extension, but in another context. There, inside (dynamically created) iframes, websites can execute JavaScript calls without OpenWPM detecting and logging this. This could also be used by websites to silently detect an OpenWPM visitor.

Further, future studies using OpenWPM should be aware that OpenWPM’s JavaScript instrumentation increases the framework’s detectability a lot.

Also, we advise the authors of future studies to always use OpenWPM in headful mode. First, general opinion on the internet is that the headless mode does not save many resources. Second, even if no display hardware is present, e.g. in cloud computations or for remote servers, one can use the functionality of Xvfb¹ or Xdummy/Xpra² to run a X server with the graphical output being rendered virtually in memory.

FP-Block. For certain spoofing configurations, *fingerprint2.js* can detect that FP-Block modified browser attributes. This infers all websites using *fingerprint2.js* can possibly identify the presence of spoofing measures for FP-Block users. This information can for example be used as additional fingerprinting attribute. The problem should be fixed as *fingerprint2.js* represents the de facto standard open source fingerprinting framework commonly used.

Although it is not the extension's aim to be robust against anti-spoofing detectors, our work showed techniques how FP-Block's spoofing methods can be improved by more advanced spoofing methods such as JavaScript proxies that complicate spoofing detection. Further, if detection of un-spoofed values via (dynamically created) iframes becomes more popular, websites could access the original value of all spoofed properties. Therefore, it is reasonable to already prevent leakage through iframes.

Browser fingerprinting. As discussed for FP-Block, also all other spoofing extensions (e.g. user agent spoofers but also academic tools) should address the problem with (dynamically created) iframes (see Section 7.1) as it allows to access the un-spoofed values. The extensions otherwise even become more fingerprintable as detected spoofing can be used as additional fingerprinting attribute, and the true values for spoofed attributes are revealed.

However, from the perspective of fingerprinting parties, it is advised to access the visitor's fingerprinting properties inside (dynamically created) iframes to increase the chances of receiving the true values.

¹X virtual frame buffer, see for example <https://kovyrin.net/2007/10/01/how-to-run-gui-programs-on-a-server-without-any-monitor/> (last visited on 04/05/2020)

²The software packet is called `xserver-xorg-video-dummy`, see for example <https://techoverflow.net/2019/02/23/how-to-run-x-server-using-xserver-xorg-video-dummy-driver-on-ubuntu/> (last visited on 04/05/2020).

Bibliography

- [Aca+13] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. “FPDetective: Dusting the Web for Fingerprinters.” In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS ’13. event-place: Berlin, Germany. New York, NY, USA: ACM, 2013, pp. 1129–1140. ISBN: 9781450324779. DOI: 10.1145/2508859.2516674. URL: <https://doi.acm.org/10.1145/2508859.2516674> (visited on 10/01/2019).
- [Chu+13] Zi Chu, Steven Gianvecchio, Aaron Koehl, Haining Wang, and Sushil Jajodia. “Blog or block: Detecting blog bots through behavioral biometrics.” en. In: *Computer Networks* 57.3 (Feb. 2013), pp. 634–646. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2012.10.005. URL: <http://www.sciencedirect.com/science/article/pii/S1389128612003593> (visited on 12/09/2019).
- [CY19] Siebren Cosijn and Nataliya Yasko. “FP-Block 2.0: preventing browser fingerprinting.” BA thesis. Open University of the Netherlands, July 19, 2019. URL: <http://www.open.ou.nl/hjo/supervision/2019-fpblock2-bsc-thesis.pdf> (visited on 12/03/2019).
- [Dol19] Rik Dolfing. *Research Internship: Towards a fingerprint surface*. Research internship. Mar. 2019. URL: <http://www.open.ou.nl/hjo/supervision/2019-rick.dolfing-msc-internship-report.pdf> (visited on 12/15/2019).
- [Eck10] Peter Eckersley. “How Unique Is Your Web Browser?” en. In: *Privacy Enhancing Technologies*. Ed. by Mikhail J. Atallah and Nicholas J. Hopper. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 1–18. ISBN: 9783642145278. DOI: 10.1007/978-3-642-14527-8_1. (Visited on 01/03/2020).
- [EN16] Steven Englehardt and Arvind Narayanan. “Online tracking: A 1-million-site measurement and analysis.” In: *Proceedings of ACM CCS 2016*. 2016. URL: http://randomwalker.info/publications/OpenWPM_1_million_site_tracking_measurement.pdf (visited on 09/16/2019).
- [Eng+15] Steven Englehardt, Chris Eubank, Peter Zimmerman, Dillon Reisman, and Arvind Narayanan. “OpenWPM: An automated platform for web privacy measurement.” Manuscript. Mar. 15, 2015. URL: https://senglehardt.com/papers/openwpm_03-2015.pdf (visited on 10/01/2019).

-
- [FE15] David Fifield and Serge Egelman. “Fingerprinting Web Users Through Font Metrics.” en. In: *Financial Cryptography and Data Security*. Ed. by Rainer Böhme and Tatsuaki Okamoto. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2015, pp. 107–124. ISBN: 9783662478547. DOI: [10.1007/978-3-662-47854-7_7](https://doi.org/10.1007/978-3-662-47854-7_7). (Visited on 01/04/2020).
- [FZW15] Amin FaizKhademi, Mohammad Zulkernine, and Komminist Weldemariam. “FPGuard: Detection and Prevention of Browser Fingerprinting.” en. In: *Data and Applications Security and Privacy XXIX*. Ed. by Pierangela Samarati. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 293–308. ISBN: 9783319208107. DOI: [10.1007/978-3-319-20810-7_21](https://doi.org/10.1007/978-3-319-20810-7_21). (Visited on 01/04/2020).
- [JKV19] Hugo Jonker, Benjamin Krumnow, and Gabry Vlot. “Fingerprint Surface-Based Detection of Web Bot Detectors.” en. In: *Computer Security – ESORICS 2019*. Ed. by Kazue Sako, Steve Schneider, and Peter Y. A. Ryan. Lecture Notes in Computer Science. Springer International Publishing, 2019, pp. 586–605. ISBN: 9783030299620. DOI: [10.1007/978-3-030-29962-0_28](https://doi.org/10.1007/978-3-030-29962-0_28). (Visited on 09/17/2019).
- [Kru+] Benjamin Krumnow, Hugo Jonker, Stefan Karsch, and Marko van Eekelen. *Fingerprint-basierte Detektion zum Untergraben von auf OpenWPM-basierenden Privatsphären- und Sicherheitsscannern*. Work in progress (available from authors).
- [Lap+19] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. “Browser Fingerprinting: A survey.” In: *arXiv:1905.01051 [cs]* (May 2019). arXiv: 1905.01051. URL: <http://arxiv.org/abs/1905.01051> (visited on 09/13/2019).
- [LBM17] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. “FPRandom: Randomizing Core Browser Objects to Break Advanced Device Fingerprinting Techniques.” en. In: *Engineering Secure Software and Systems*. Ed. by Eric Bodden, Mathias Payer, and Elias Athanasopoulos. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 97–114. ISBN: 9783319621050. DOI: [10.1007/978-3-319-62105-0_7](https://doi.org/10.1007/978-3-319-62105-0_7). (Visited on 01/04/2020).
- [Le +19] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. “Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation.” In: *Proceedings of the 26th Annual Network and Distributed System Security Symposium*. NDSS 2019. Feb. 2019. DOI: [10.14722/ndss.2019.23386](https://doi.org/10.14722/ndss.2019.23386). (Visited on 02/17/2020).

- [LRB15] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. “Mitigating Browser Fingerprint Tracking: Multi-level Reconfiguration and Diversification.” In: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ISSN: 2157-2321. May 2015, pp. 98–108. DOI: [10.1109/SEAMS.2015.18](https://doi.org/10.1109/SEAMS.2015.18). (Visited on 01/05/2020).
- [MS12] Keaton Mowery and Hovav Shacham. *Pixel Perfect: Fingerprinting Canvas in HTML 5*. 2012. URL: <https://hovav.net/ucsd/papers/ms12.html> (visited on 01/03/2020).
- [Mul+13] Martin Mulazzani, Philipp Reschl, Markus Huber, Manuel Leithner, Sebastian Schrittwieser, and Edgar Weippl. *Fast and Reliable Browser Identification with JavaScript Engine Fingerprinting*. Presented at Web 2.0 Security & Privacy 2013. SBA Research, 2013. URL: <https://publications.sba-research.org/publications/jsfingerprinting.pdf> (visited on 01/04/2020).
- [Nik+13] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. “Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting.” In: *2013 IEEE Symposium on Security and Privacy*. ISSN: 1081-6011. May 2013, pp. 541–555. DOI: [10.1109/SP.2013.43](https://doi.org/10.1109/SP.2013.43).
- [NJJ15] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. “PriVaricator: Deceiving Fingerprinters with Little White Lies.” In: *Proceedings of the 24th International Conference on World Wide Web*. WWW ’15. Florence, Italy: International World Wide Web Conferences Steering Committee, 2015, pp. 820–830. ISBN: 9781450334693. DOI: [10.1145/2736277.2741090](https://doi.org/10.1145/2736277.2741090). URL: <https://doi.org/10.1145/2736277.2741090> (visited on 01/04/2020).
- [Ole+16] Łukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. “The Leaking Battery.” en. In: *Data Privacy Management, and Security Assurance*. Ed. by Joaquin Garcia-Alfaro, Guillermo Navarro-Arribas, Alessandro Aldini, Fabio Martinelli, and Neeraj Suri. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 254–263. ISBN: 9783319298832. DOI: [10.1007/978-3-319-29883-2_18](https://doi.org/10.1007/978-3-319-29883-2_18). (Visited on 01/04/2020).
- [RK18] Philip Raschke and Axel Küpper. “Uncovering Canvas Fingerprinting in Real-Time and Analyzing Its Usage for Web-Tracking.” en. In: *Workshops der INFORMATIK 2018 - Architekturen, Prozesse, Sicherheit und Nachhaltigkeit*. Ed. by Christian Czarnecki, Carsten Brockmann, Eldar Sultanow, Agnes Koschmider, and Annika Selzer. Köllen Druck+Verlag GmbH, 2018, pp. 97–108. ISBN: 9783885796794. URL: <http://dl.gi.de/handle/20.500.12116/17237> (visited on 04/01/2020).

- [SLG19] Michael Schwarz, Florian Lackner, and Daniel Gruss. “JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits.” en. In: *Proceedings 2019 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2019. ISBN: 9781891562556. DOI: 10.14722/ndss.2019.23155. URL: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_01B-4_Schwarz_paper.pdf (visited on 09/15/2019).
- [SN17] Oleksii Starov and Nick Nikiforakis. “XHOUND: Quantifying the Fingerprintability of Browser Extensions.” In: *2017 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA, USA: IEEE, May 2017, pp. 941–956. ISBN: 9781509055333. DOI: 10.1109/SP.2017.18. URL: <http://ieeexplore.ieee.org/document/7958618/> (visited on 01/03/2020).
- [SPK16] *I’m not a human: Breaking the Google reCAPTCHA*. Black Hat Asia 2016. Black Hat. 2016. URL: <https://www.blackhat.com/docs/asia-16/materials/asia-16-Sivakorn-Im-Not-a-Human-Breaking-the-Google-reCAPTCHA-wp.pdf> (visited on 01/04/2020).
- [Sta+19] Oleksii Starov, Pierre Laperdrix, Alexandros Kapravelos, and Nick Nikiforakis. “Unnecessarily Identifiable: Quantifying the Fingerprintability of Browser Extensions Due to Bloat.” In: *The World Wide Web Conference. WWW ’19*. San Francisco, CA, USA: Association for Computing Machinery, 2019, pp. 3244–3250. ISBN: 9781450366748. DOI: 10.1145/3308558.3313458. (Visited on 01/05/2020).
- [Sto+17] Grant Storey, Dillon Reisman, Jonathan Mayer, and Arvind Narayanan. “The Future of Ad Blocking: An Analytical Framework and New Techniques.” In: *arXiv:1705.08568 [cs]* (May 2017). URL: <http://arxiv.org/abs/1705.08568> (visited on 01/21/2020).
- [TJM15] Christof Ferreira Torres, Hugo Jonker, and Sjouke Mauw. “FP-Block: Usable Web Privacy by Controlling Browser Fingerprinting.” en. In: *Computer Security – ESORICS 2015*. Ed. by Günther Pernul, Peter Y A Ryan, and Edgar Weippl. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 3–19. ISBN: 9783319241777. DOI: 10.1007/978-3-319-24177-7_1. (Visited on 09/17/2019).
- [Ung+13] Thomas Unger, Martin Mulazzani, Dominik Frühwirt, Markus Huber, Sebastian Schrittwieser, and Edgar Weippl. “SHPF: Enhancing HTTP(S) Session Security with Browser Fingerprinting.” In: *2013 International Conference on Availability, Reliability and Security*. ISSN: null. Sept. 2013, pp. 255–261. DOI: 10.1109/ARES.2013.33. (Visited on 01/04/2020).

- [Vas+18] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. “FP-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies.” In: *Proceedings of the 27th USENIX Security Symposium*. Baltimore, United States, Aug. 2018. URL: <https://hal.inria.fr/hal-01820197> (visited on 12/10/2019).
- [Vlo18] Gabry Vlot. “Automated data extraction; what you see might not be what you get.” MA thesis. Open University of the Netherlands, July 5, 2018. URL: <http://www.open.ou.nl/hjo/supervision/2018-g.vlot-msc-thesis.pdf> (visited on 11/12/2019).

Appendices

A. Example fingerprint

There is no exhaustive list of properties and behavior that can be used for browser fingerprinting. To show an example of an actual fingerprint, [Table A.1](#) lists the information *fingerprint2.js* collected for a Google Chrome instance on a specific machine.

Table A.1.: Fingerprint of a Google Chrome instance, generated by *fingerprint2.js*. Very long hashed values are truncated by “...”.

Property	Value(s)
userAgent	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.108 Safari/537.36
webdriver	not available
language	en-US
colorDepth	24
deviceMemory	8
hardwareConcurrency	8
screenResolution	1280, 1024
availableScreenResolution	1280, 1024
timezoneOffset	-60
timezone	Europe/Berlin
sessionStorage	true
localStorage	true
indexedDb	true
addBehavior	false
openDatabase	true
cpuClass	not available
platform	Linux x86_64

Table A.1 (continued): Fingerprint of a Google Chrome instance, generated by *fingerprint2.js*. Very long hashed values are truncated by “...”.

Property	Value(s)
plugins	Chrome PDF Plugin, Portable Document Format, application/x-google-chrome-pdf, pdf, Chrome PDF Viewer, application/pdf, pdf, Native Client, application/x-nacl, application/x-pnacl
canvas	canvas winding:yes, canvas fp:data:image/png;base64, iVBORw...
webgl	data:image/png;base64, iVBORw...
webglVendorAndRenderer	Intel Open Source Technology Center~Mesa DRI Intel(R) UHD Graphics (Whiskey Lake 3x8 GT2)
adBlock	false
hasLiedLanguages	false
hasLiedResolution	false
hasLiedOs	false
hasLiedBrowser	false
touchSupport	0, false, false
fonts	Andale Mono, Arial, Arial Black, Comic Sans MS, Courier, Courier New, Georgia, Helvetica, Impact, Tahoma, Times, Times New Roman, Trebuchet MS, Verdana
audio	124.04344884395687

B. Experiment environment

Our first idea was to analyze OpenWPM’s detectability inside a Ubuntu virtual machine. However, we wanted to exclude the virtualization as source of disruption in our measurements. Therefore, if not explicitly indicated, all experiments are performed on the dedicated machine described in [Table B.1](#).

Table B.1.: Specification of the environment for the [How stable is OpenWPM’s fingerprint surface?](#) chapter.

Type	Component	Description
Hardware	CPU	Intel® Core™ i5-2430M CPU 2.40GHz x 4
	Architecture	64-bit
	RAM	4 GB
Software	OS	Ubuntu 18.04.3 LTS x86_64
	Regular Firefox (baseline)	70.0
	Firefox unbranded	70.0
	Geckodriver	0.24.0 (2019-01-28)
	Python	3.6.9
	Selenium WebDriver (Python)	3.141.0

C. Firefox extensions: Page-, content- and background-scripts

When dealing with Firefox extensions, JavaScript code is executed at three locations:

1. Background scripts,
2. content scripts,
3. page scripts.

The three script types run in different scopes that, per default, are separated from each other such that modifications remain in one script. Their relation is depicted in [Figure C.1](#). When modifying the DOM in a Firefox extension, it is important to propagate the modification to the page scope that website scripts get to see. An example follows.

If a user opens a website, the browser loads the requested web page and its resources such as HTML or JavaScript. Scripts provided by websites are called page scripts. They have access to the page DOM.

Now, suppose a Firefox extensions wants to modify the page DOM, for example overwrite `document.title`. If it injects a content script that overwrites `document.title` into the web page, nothing seems to happen. This is caused by content scripts been considered more privileged code (Firefox extensions are under the user's control while websites can contain arbitrary JavaScript). In this regard, Firefox introduced the Xray vision¹ that basically separates two versions of the DOM such that DOM modifications by page scripts are invisible for content scripts and vice versa.

Background scripts have no access to the DOM at all (they contain functionality that an extension runs the whole time, mainly extension logic that needs to be preserved across different pages).

So, the first attempt of a content script well overwrite `document.title`, but only in the scope of the content script itself. Website/page scripts still saw an unmodified DOM. However, with explicit commands, content scripts of an extension can share objects to the page DOM (`exportFunction()` respectively `cloneInto()`) and also access object of it (`.wrappedJSObject`).

¹https://developer.mozilla.org/en-US/docs/Mozilla/Tech/Xray_vision (last visited on 02/03/2020)

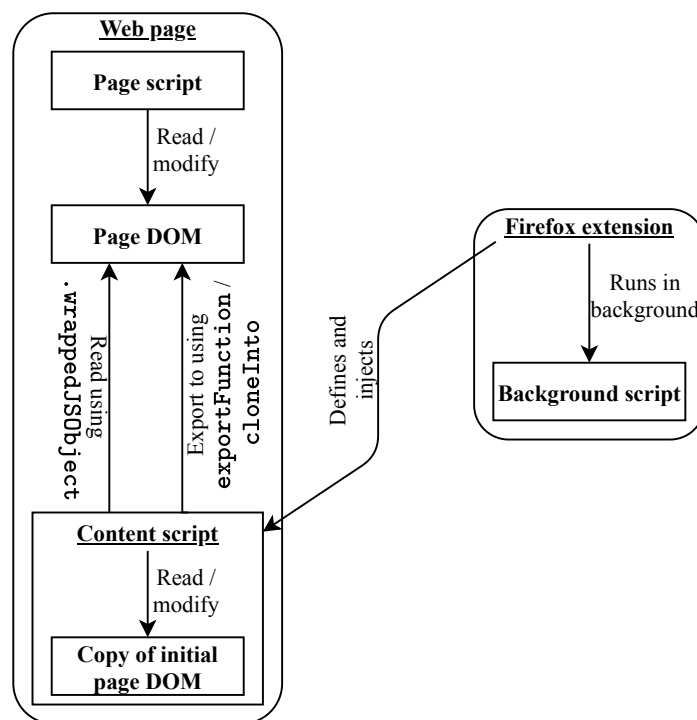


Figure C.1.: Simplified overview of JavaScript execution for Firefox extensions. Background- and content-scripts are controlled by the extension, while page scripts are already contained in a web page.

D. Investigation of divergent window-sizes of OpenWPM

In headful mode, OpenWPM's window size deviates from that of regular Firefox (see [Table 4.2](#)). This will be investigated in this section. We will not further investigate headless mode, where deviations of the window size are caused by the availability of the full screen resolution.

Comparing the JavaScript template attack measurements captured in [Chapter 4](#), the main diverging JavaScript properties are:

- `window.innerWidth`: Width of the browser viewport, that is the part where the actual content is rendered, including possible vertical scrollbar;
- `window.innerHeight`: Height of the browser viewport, that is the part where the actual content is rendered, including possible horizontal scrollbar;
- `window.outerWidth`: Width of the whole browser window;
- `window.outerHeight`: Height of the whole browser window, but without the title bar of Ubuntu that contains the window title “Mozilla Firefox” respectively “Nightly” (also see [Figure 5.1](#)).

All other window-size related JavaScript properties are linked to these or dependent on the position of the window that can change.

To find the exact source of the deviations, we analyze the browser sizes in a new Ubuntu 18.04 LTS virtual machine with a screen resolution of 1366x768. There, we use the JavaScript template attack framework to measure the JavaScript environment of a regular Firefox instance and OpenWPM in order to compare them.

We notice a behavioral difference between the startup of Firefox and OpenWPM.

- Behavior of regular Firefox:
 - Per default opens as maximized window with a full program dimension of 1293x741.
 - The standard smaller window (by moving the maximized window it becomes smaller automatically) has a dimension of 1156x691.

- Behavior of OpenWPM:
 - Per default opens as full-size window (not maximized) with a full program dimension of 1293x741.
 - If manually maximized, the program dimension is still 1293x741, but in JavaScript the `window.innerHeight` and `window.outerHeight` properties are incremented by two pixels. This originates from 2 pixels less shadow effects by the Ubuntu skin if the window is (truly) maximized and not only of full screen size.
 - Only after manual maximizing and de-maximizing the window, it resizes to a standard smaller window. Then, the program dimension is 1156x662, but it is not preserved across spawned OpenWPM instances.

Conclusions. In this experiment, Firefox and OpenWPM do not differ in their resolution if both windows are maximized, but OpenWPM does not maximize its window by default but only spawns it as full-size window. Therefore, a difference of a few pixels is measured, caused by window shadow effects.

Furthermore, this experiment could not reproduce the larger deviations found in [Chapter 4](#). It is suspected that using the test system's browser before running the experiment changed its behavior to spawn Firefox not maximized but in a standard window. During the experiment, we also encountered Ubuntu to not always handle the window-sizes the same way. Sometimes, windows suddenly spawned maximized and directly thereafter not.

So, more in general, default spawned windows do not necessarily share the same window dimensions across programs. Further, all kinds of windows effects, such as shadows, can lead to differences.

E. Spoofing detection in iframes

During research for JavaScript spoofing and methods that still allow to reveal the original, un-spoofed value, we identified advanced countermeasures by the extension *User-Agent Switcher* by Alexander Schlarb. It ships along with a list of different tricks that can be used to potentially access the original value of JavaScript properties.¹ They are based on the use of static and dynamic iframes. On their basis, we crafted a slightly modified test page to check whether the four other popular Firefox extensions that spoof the browser’s user agent (investigated in Section 5.2) can deal with iframes in another way. Additionally, FP-Block is tested to see whether it can be improved.

Setup. For each extension, the test page is visited with Firefox Nightly 70.0² to determine the `navigator.userAgent` property. It is checked for the `userAgent` because it is modified by all extensions while no extension modifies `webdriver`. Before each measurement, the browser is restarted and all browser history is cleared. Further, if not denoted otherwise, the extensions run in standard settings, but with a manually chosen user agent profile. The extension *Chameleon* is tested in both standard setting and with activated script injection, where the spoofing measures are more advanced.

E.1. Results

All extensions succeed in spoofing the property when it is accessed the standard way by querying `window.navigator.userAgent` in page scope. However, inside both static and dynamic iframes, the `userAgent` property is not spoofed immediately but delayed such that the original value can be extracted first. This is caused by Firefox that only allows content script injection to empty iframes on a later stage than for regular pages³.

The results of the detailed iframe test cases are depicted in Table E.1. The table shows that all tested extensions leak the unaltered user agent in some test cases. In the following, the table is discussed in more detail.

¹Alexander Schlarb’s test list, which our test page is based up on, can be found at https://gitlab.com/ntninja/user-agent-switcher/-/issues/9#note_194036885 (last visited on 03/23/2020).

²In standard settings, besides deactivated automatic Firefox updates and deactivated Firefox extension signing check.

³https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/content_scripts (last visited on 01/30/2020)

Table E.1.: Test under which circumstances it is still possible to read out the un-spoofed `navigator.userAgent` attribute of five popular user agent spoofing extensions and FP-Block 2.0. The list of access methods is based on Alexander Scharb’s tests.

JavaScript access method	Firefox spoofing extension ¹						
	Chameleon (standard)	Chameleon (script injection)	Random User-Agent	UAS by Linder	UAS by Alexander Scharb	UAS and Manager	FP-Block 2.0 ²
Standard access	–	–	–	–	–	–	–
Static iframe (<code>window[0]</code>)	●	–	●	○	○	●	○
Static iframe (<code>window.frames[0]</code>)	●	–	●	○	–	●	○
Static iframe (<code>.contentWindow</code>)	●	–	●	○	–	●	○
Dynamic iframe (<code>window[1]</code>)	●	○	●	●	○	○	●
Dynamic iframe (<code>window.frames[1]</code>)	●	○	●	●	–	○	●
Dynamic iframe (<code>.contentWindow</code>)	●	○	●	●	–	○	●
Dynamic nested iframe (<code>.contentWindow</code>)	●	●	●	●	–	○	●

- `navigator.userAgent` successfully spoofed respectively the un-spoofed value could not be read.
- Un-spoofed value of `navigator.userAgent` could be read, but only if this access happened immediately (and not in a later stadium in JavaScript).
- Un-spoofed value of `navigator.userAgent` could be read, independent of access time.

¹ The extensions are the same as in Table 5.2 (page 26). See there for their corresponding links to Mozilla’s extension archive.

² FP-Block 2.0’s main focus is not to be robust against advanced JavaScript methods to receive un-spoofed values, which should be kept in mind when comparing it against other extensions.

FP-Block 2.0's situation. It should be highlighted that FP-Block's focus lies on generating and leveraging consistent and complete fingerprint profiles. Therefore, FP-Block is not made robust against fingerprint spoofing detectors (that are specifically tailored to recognize spoofing visitors). Nevertheless, the test results allow improving FP-Block's manner of JavaScript spoofing to improve its capabilities.

So what do the results show? Unsurprisingly, spoofing succeeds for both immediate and delayed access if the property is accessed in the standard manner. But for immediately accessed static as well dynamic iframes, the value is not spoofed at all. For delayed access, spoofing succeeds at least for static iframes. This means it is possible to receive the original, un-spoofed value using the appropriate tricks. Therefore, we suggest improving FP-Block's method of spoofing, albeit that is not FP-Block's main focus.

Standard tools. Two extensions attract attention because they only successfully spoof for standard access: *Chameleon (standard)* and *Random User-Agent*. They do not perform more advanced JavaScript attribute overwriting such as injecting a content script. It should be noted *Chameleon* has a more advanced mode of operation measured separately.

Further, depending on the spoofing method, extensions fail in different test cases. Each “○” in Table E.1 denotes a case where only immediate access has a problem and delayed access not. So overall, delayed access of the user agent property performs better than immediate access. Several extensions inject JavaScript content scripts to each page to allow for more advanced spoofing. With delayed access, these scripts had enough time to do their work. By design, it is not possible to enforce that the scripts injected by extensions are always loaded before page scripts.

Moreover, spoofing measures taken by the extensions sometimes seem to work solely for dynamically created iframes (such as for *UAS and Manager*), while other extensions, such as *UAS by Linder*, only deal with static iframes. When the user agent is instantly accessed, all but *UAS by Alexander Schlarb* are not capable to spoof it inside dynamically created iframes.

The two best performing extensions are *Chameleon (with script injection)* and *UAS by Alexander Schlarb*. They both inject an advanced content script to also spoof the user agent for iframes in most cases. However, even they are not capable of spoofing the user agent for all access methods. A simple combination of them is unfortunately not possible.

Alexander Schlarb's extension contains the most sophisticated countermeasures. It monitors all added iframes and triggers a previously defined property such that they are spoofed immediately, too. Unfortunately, access via `window[0]/window.frames[0]` still cannot be prevented as only one array index at a time can be overwritten and the number of indices is unlimited.

Conclusions. First, the analysis shows none of the tested standard extensions is capable of spoofing JavaScript properties perfectly. Not astonishing is the fact that *User-Agent Switcher by Alexander Schlarb* turned out to be one of the better extensions, as the tested access methods originate from the author of this extension.

Second, instantly accessed dynamic iframes pose a problem to the majority of the extensions. Therefore, we propose that browser fingerprinting in general should always query the value of JavaScript properties inside dynamic iframes to detect spoofing and/or get their original values.

F. Technical details on spoofing the webdriver property

Here, the JavaScript code of the four methods presented in [Section 6.3](#) is given and, if applicable, further technical details are discussed. Their strengths and weaknesses are reviewed in [Section 6.4](#).

To prevent complexity, we use `window.eval(...)` at some places to avoid bothering with page and content script scopes. This command directly executes the given JavaScript code in the page scope.

The four methods work as follows:

1. `Object.defineProperty` method:

```
1 window.eval("Object.defineProperty(navigator, 'webdriver',  
  ↪ {enumerable: true, value: false});");
```

This is the way currently proposed to redefine properties¹.

Alternatively, one can use:

```
1 window.eval("Object.defineProperty(navigator, 'webdriver', {  
  ↪ get() {return false;} });");
```

As with `enumerable: false`, this alternative version is not measurable using `Object.keys(navigator)`. However, `window.navigator._length` is still incremented and `webdriver` is missing completely when `navigator` is enumerated (e.g. printed in the console or read by *BrowserBasedBotFP*).

2. `Object.prototype.__defineGetter__` method:

```
1 let funcPageScope = exportFunction(() => {  
2   return false;  
3 }, window.wrappedJSObject);
```

¹See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/__defineGetter__ (last visited on 01/30/2020)

F. Technical details on spoofing the webdriver property

```
4 Object.prototype.__defineGetter__.call(window.navigator.wrapped_
  ↪  JavaScript, "webdriver",
  ↪  funcPageScope);
```

This is the deprecated method used prior to `Object.defineProperty`. It is still supported by all major browsers¹.

3. `Object.setPrototypeOf` method:

```
1 window.eval("const newProto = Object.getPrototypeOf(navigator);
  ↪  Object.defineProperty(newProto, 'webdriver', {enumerable:
  ↪  true, value: false}); Object.setPrototypeOf(navigator,
  ↪  newProto);");
```

This sets a prototype on `navigator` that overshadows the actual value of the `webdriver` property.

4. JavaScript proxy method:

A JavaScript proxy replaces the original `navigator` object. Then, it resembles the behavior of the original `navigator` object except spoofing another value for the `navigator.webdriver` property. The following code² additionally deals with the `iframe` problem discussed in [Section 7.1](#).

Note the code contains an `Object.prototype.__defineGetter__` statement to overwrite `navigator` with the JavaScript proxy. In [Section 6.4](#), `__defineGetter__` caused detection in form of a new key on `navigator`. The same does not happen here, so no new `navigator` key is added to `window` because `navigator` is already a key of the `window` object by default.

Lines 45-47 are responsible for the 20 `window.navigator` functions that lose their names.

```
1  /**
2   * This code to spoof values of the `window.navigator` object using a
  ↪  JavaScript proxy is based on:
3   * User Agent Switcher, copyright © 2017 - 2019 Alexander Schlarb
  ↪  (https://gitlab.com/ntninja)
4   * For the used part see:
  ↪  https://gitlab.com/ntninja/user-agent-switcher/blob/6aacc15ed6651317776f7\_abb3a85d6f34fca254/content/override-navigator-data.js
5   *
```

²The code is also available on GitHub, see [Chapter 1](#).

```

6   * This program is free software: you can redistribute it and/or modify it
   ↪ under the terms of the GNU General Public License as published by the
   ↪ Free Software Foundation, either version 3 of the License, or (at your
   ↪ option) any later version.
7   *
8   * You should have received a copy of the GNU General Public License along
   ↪ with this program. If not, see <http://www.gnu.org/licenses/>.
9   */
10
11  /**
12   * Set of all object that we have already proxied to prevent them from being
   ↪ proxied twice.
13   */
14  let proxiedObjects = new Set();
15
16  /**
17   * Convenience wrapped around `cloneInto` that enables all possible cloning
   ↪ options by default.
18   */
19  function cloneIntoFull(value, scope) {
20      return cloneInto(value, scope, {
21          cloneFunctions: true,
22          wrapReflectors: true
23      });
24  }
25
26  /**
27   * Spoof `navigator` by overwriting the `navigator` of the given (content
   ↪ script scope) `window` object with a proxy, if applicable.
28   */
29  function spoofNavigator(window) {
30      if (!(window instanceof Window)) { // Not actually a window object
31          return window;
32      }
33
34      let origNavigator = window.navigator.wrappedJSObject; // `navigator` of
   ↪ the page scope
35      if (proxiedObjects.has(origNavigator)) { // Window was already shadowed
36          return window;
37      }
38
39      let spoofedGet_PageScope = cloneIntoFull({
40          get: (target, prop, receiver) => {
41              if (prop === "webdriver") {
42                  return false;
43              } else {
44                  let value = Reflect.get(origNavigator, prop);

```

F. Technical details on spoofing the webdriver property

```
45     if(typeof(value) === "function") { // Bind functions like
      ↪ `navigator.javaEnabled()` to the original object in the
      ↪ page scope to allow them to execute
46     let boundFunc = Function.prototype.bind.call(value,
      ↪ origNavigator); // `value` is used as `this` to call
      ↪ the `bind` function that creates a copy of
      ↪ `Function.prototype` that always runs in the `this`
      ↪ context `origNavigator`
47     value = cloneIntoFull(boundFunc, window.wrappedJSObject);
48   }
49   return value;
50 }
51 }
52 }, window.wrappedJSObject); // The `get` function, defined in privileged
  ↪ code (that is here in the extension / content script), is cloned into
  ↪ the target scope (that is the web page / `window.wrappedJSObject`)
  ↪ and thus accessible there. The return value is the reference to the
  ↪ cloned object in the defined scope.
53
54 let origProxy = window.wrappedJSObject.Proxy;
55 let navigatorProxy = new origProxy(origNavigator, spoofedGet_PageScope);
56
57 proxiedObjects.add(origNavigator);
58
59 let returnFunc_PageScope = exportFunction(() => {
60   return navigatorProxy;
61 }, window.wrappedJSObject);
62 // Using `__defineGetter__` here our function gets assigned the correct
  ↪ name of `get navigator`. Additionally its property descriptor has no
  ↪ `set` function and will silently ignore any assigned value. - This
  ↪ exact configuration is not achievable using `Object.defineProperty`.
63 Object.prototype.__defineGetter__.call(window.wrappedJSObject,
  ↪ "navigator", returnFunc_PageScope);
64 return window;
65 }
66
67 /**
68  * Override `navigator` with the given data on the given page scoped `window`
  ↪ object if applicable. This will convert the given `window` object to
  ↪ being content-script scoped after checking whether it can be converted at
  ↪ all or is just a restricted accessor that does not grant access to
  ↪ anything important.
69  */
70 function spoofNavigatorFromPageScope(unsafeWindow) {
71   if(!(unsafeWindow instanceof Window)) {
72     return unsafeWindow; // Not actually a window object
73   }
74
75   try {
```

```

76     unsafeWindow.navigator; // This will throw if this is a cross-origin
    ↪ frame
77
78     let windowObj = cloneIntoFull(unsafeWindow, window);
79     return spoofNavigator(windowObj).wrappedJSObject;
80 } catch(e) {
81     if(e instanceof DOMException && e.name == "SecurityError") {
82         // Ignore error created by accessing a cross-origin frame and
    ↪ just return the restricted frame (`navigator` is inaccessible
    ↪ on these so there is nothing to patch)
83         return unsafeWindow;
84     } else {
85         throw e;
86     }
87 }
88 }
89
90
91 spoofNavigator(window);
92
93 // Use some prototype hacking to prevent access to the original `navigator`
    ↪ through the IFrame leak
94 const IFRAME_TYPES = Object.freeze([HTMLFrameElement, HTMLIFrameElement]);
95 for(let type of IFRAME_TYPES) {
96     // Get reference to contentWindow & contentDocument accessors into the
    ↪ content script scope
97     let contentWindowGetter = Reflect.getOwnPropertyDescriptor(
98         type.prototype.wrappedJSObject, "contentWindow"
99     ).get;
100    contentWindowGetter = cloneIntoFull(contentWindowGetter, window);
101    let contentDocumentGetter = Reflect.getOwnPropertyDescriptor(
102        type.prototype.wrappedJSObject, "contentDocument"
103    ).get;
104    contentDocumentGetter = cloneIntoFull(contentDocumentGetter, window);
105
106    // Export compatible accessor on the property that patches the navigator
    ↪ element before returning
107    Object.prototype.__defineGetter__.call(type.prototype.wrappedJSObject,
    ↪ "contentWindow",
108        exportFunction(function () {
109            let contentWindow = contentWindowGetter.call(this);
110            return spoofNavigatorFromPageScope(contentWindow);
111        }, window.wrappedJSObject)
112    );
113    Object.prototype.__defineGetter__.call(type.prototype.wrappedJSObject,
    ↪ "contentDocument",
114        exportFunction(function () {
115            let contentDocument = contentDocumentGetter.call(this);
116            if(contentDocument !== null) {

```

F. Technical details on spoofing the webdriver property

```
117         spoofNavigatorFromPageScope(contentDocument.defaultView);
118     }
119     return contentDocument;
120     }, window.wrappedJSObject)
121 );
122 }
123
124 // Asynchronously track added IFrame elements and trigger their prototype
125 ↪ properties defined above to ensure that they are patched (This is a
126 ↪ best-effort workaround for us being unable to *properly* fix the
127 ↪ `window[0]` case.)
128 let patchNodes = (nodes) => {
129     for(let node of nodes) {
130         let isNodeFrameType = false;
131         for(let type of IFRAME_TYPES) {
132             if(isNodeFrameType = (node instanceof type)){ break; }
133         }
134         if(!isNodeFrameType) {
135             continue;
136         }
137         node.contentWindow;
138         node.contentDocument;
139     }
140 };
141 let observer = new MutationObserver((mutations) => {
142     for(let mutation of mutations) {
143         patchNodes(mutation.addedNodes);
144     }
145 });
146 observer.observe(document.documentElement, {
147     childList: true,
148     subtree: true
149 });
150 patchNodes(document.querySelectorAll("frame,iframe"));
```

G. Effects of webdriver detection

This section shows further examples caused by `webdriver` detection encountered in Chapter 7.

Access Denied

You don't have permission to access "http://www.alitalia.com/it_it/offerte/tutte-le-offerte/offer-detail.html?" on this server.
Reference #18.7fd86b68.1585139046.605f5686

Figure G.1.: Blockage on `alitalia.com`.

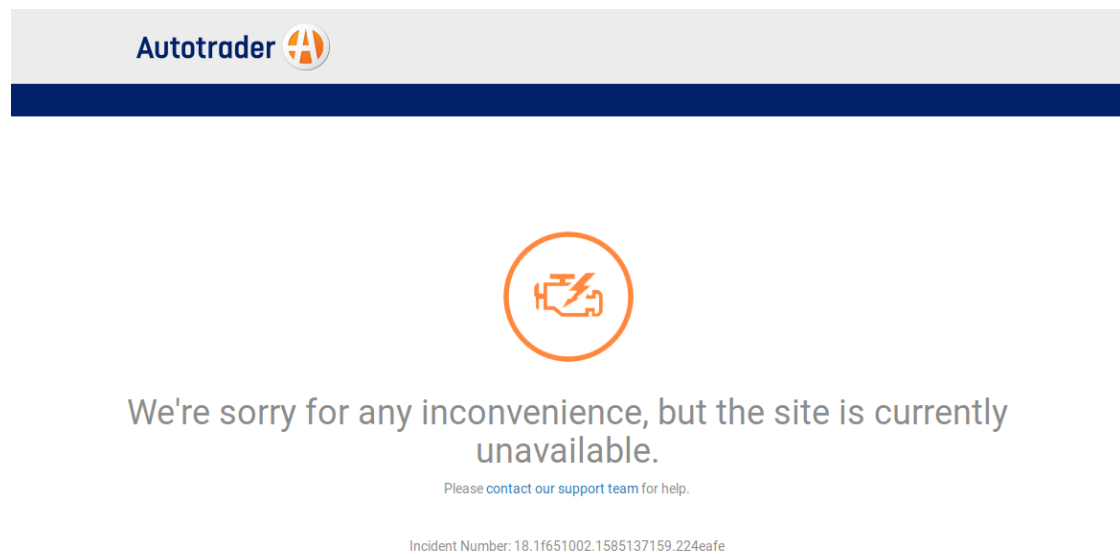


Figure G.2.: Blockage on `autotrader.com`.

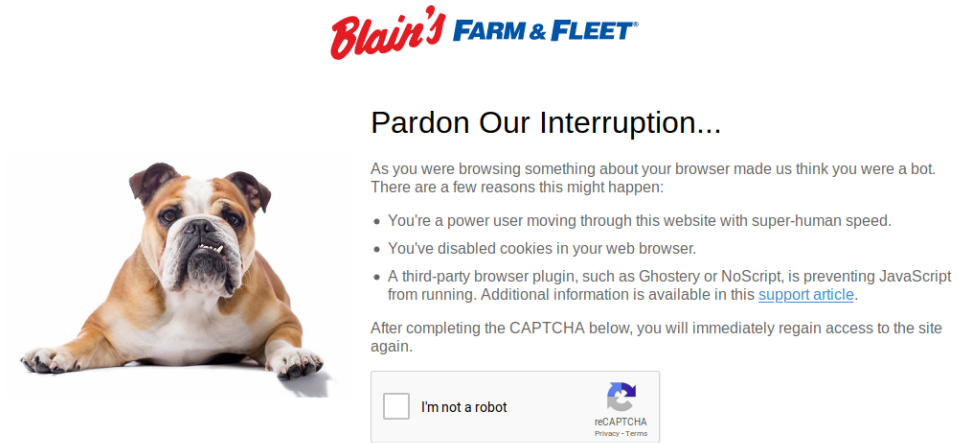


Figure G.3.: Blockage on farmandfleet.com.

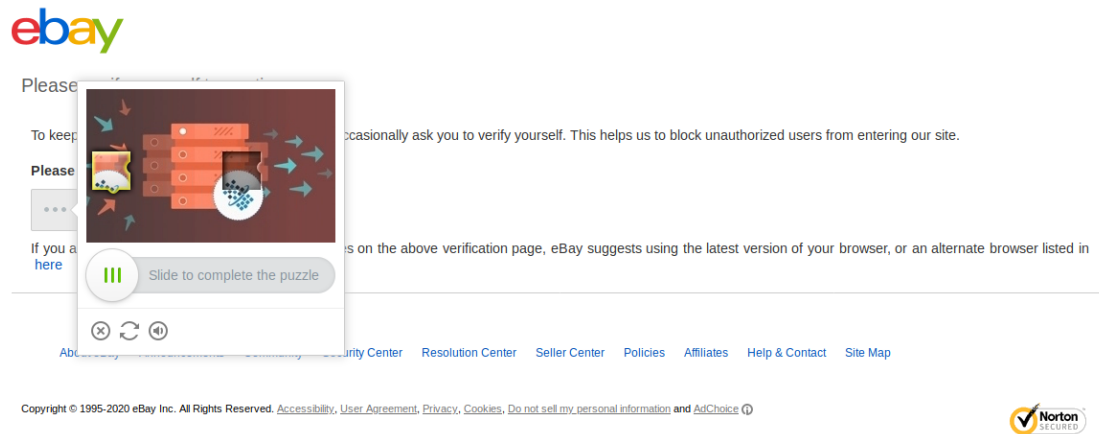


Figure G.4.: CAPTCHA for login on ebay.com.