FP-Crawlers: Studying the Resilience of Browser Fingerprinting to Block Crawlers

Antoine Vastel, Walter Rudametkin, Romain Rouvoy, Xavier Blanc

FP-CRAWLERS: Studying the Resilience of Browser Fingerprinting to Block Crawlers

Antoine Vastel
Datadome
antoine.vastel@datadome.co

Walter Rudametkin
Univ. Lille / Inria
walter.rudametkin@univ-lille.fr

Romain Rouvoy
Univ. Lille / Inria / IUF
romain.rouvoy@univ-lille.fr

Xavier Blanc
Univ. Bordeaux / IUF
xavier.blanc@u-bordeaux.fr

***Abstract*—Data available on the Web, such as financial data or public reviews, provides a competitive advantage to companies able to exploit them. Web crawlers, a category of bot, aim at automating the collection of publicly available Web data. While some crawlers collect data with the agreement of the websites being crawled, most crawlers do not respect the terms of service. CAPTCHAs and approaches based on analyzing series of HTTP requests classify users as humans or bots. However, these approaches require either user interaction or a significant volume of data before they can classify the traffic.**

In this paper, we study browser fingerprinting as a crawler detection mechanism. We crawled the Alexa top 10K and identified 291 websites that block crawlers. We show that fingerprinting is used by 93 (31.96%) of them and we report on the crawler detection techniques implemented by the major fingerprinters. Finally, we evaluate the resilience of fingerprinting against crawlers trying to conceal themselves. We show that although fingerprinting is good at detecting crawlers, it can be bypassed with little effort by an adversary with knowledge on the fingerprints collected.

I. INTRODUCTION

A majority of the web's traffic is due to crawlers [32], [41], which are programs that explore websites to extract data. While such crawlers provide benefits to the websites they crawl—*e.g.*, by increasing visibility from search engines—other crawlers' sole intent is to scrape valuable data to provides competitive advantages. Some businesses crawl their competitors' websites, to adjust their pricing strategy, while others copy, republish and monetize content without permission. The legal and moral issues of crawling have been discussed [6], [45], and companies have sued [12] and won against crawlers [61].

To protect from undesired crawling, most websites host a `robots.txt` file that specifies the pages that can be crawled and indexed. However, there is no mechanism to force malicious crawlers to respect it. CAPTCHAs [59] are popular to detect malicious crawlers, but progress in automatic image and audio recognition, as well as for-profit crowdsourcing services [49], [1], [10], mean they can be bypassed [7]. Other techniques rely on analyzing the sequence of requests sent by the client. Rate limiting techniques [51], [52], [5], [60] analyze features, such as the number of requests or pages loaded, to classify the client as human or crawler, while more advanced techniques extract features from time series [33].

Browser fingerprinting is a less studied approach to detect crawlers. Fingerprints are attributes that characterize a user's device and browser. While fingerprinting has been studied for tracking [20], [58], [43], [2], [22], [36], [42], it has received less attention for security purposes [9], [3], [44], [54]. Browser fingerprinting addresses some of the weaknesses of state-of-the-art crawler detection techniques, such as:

- Contrary to CAPTCHAs, fingerprinting does not require user interaction,
- Contrary to methods based on HTTP requests or time series, fingerprinting requires a single request to decide whether a client is a crawler.

This work aims to improve the understanding of browser fingerprinting for crawler detection. We crawled the Alexa Top 10K websites to answer the following questions:

A. What ratio of websites have adopted browser fingerprinting for crawler detection?
B. What are the key detection techniques implemented by major commercial fingerprinters?
C. How resilient is fingerprinting against adversaries that alters fingerprints to escape detection?

The contributions of our work are:

1) **Adoption of fingerprinting** as a crawler detection mechanism is studied in the wild by analyzing the scripts present on the websites of the Alexa's Top 10K;
2) **Fingerprinting techniques:** we deobfuscate the 4 most common fingerprinting scripts that detect crawlers. We present the key attributes and how they reveal crawlers;
3) **Resilience to adversaries:** we create crawlers with distinct fingerprints to evaluate the resilience of fingerprinting against adversaries that try to evade detection. We measure the effort required and show that fingerprinting is good at detecting crawlers that evade state-of-the-art techniques, but can still be bypassed by knowledgeable adversaries.

An overview of FP-CRAWLERS, including our contributions and the paper's structure, is depicted in Figure 1. In particular, Section III crawls the Alexa's Top 10K to find websites that use fingerprinting to detect crawlers. Section IV explains the key techniques they use. Section V evaluate the resilience of fingerprinting against adversarial crawlers.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.23xxx
www.ndss-symposium.org

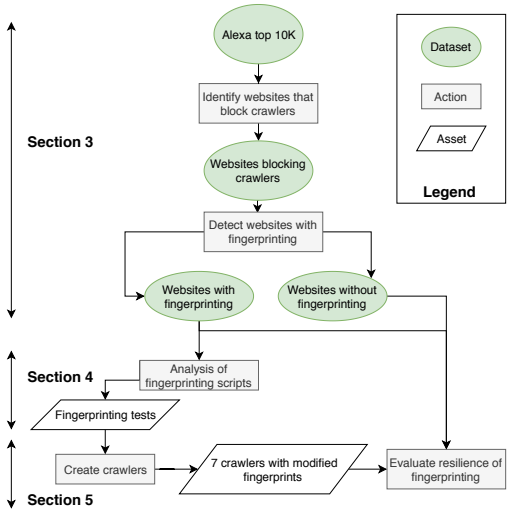

Fig. 1: Overview of FP-CRAWLERS.

II. RELATED WORK

A. Crawler & Bot Detection Techniques

Traffic shape analysis. Jacob *et al.* [33] propose a system to detect crawlers. Contrary to other approaches that rely on the concept of user sessions [51], [52], [5], [60], they decompose time series into a trend, a season and a noise component, and extract features such as the *sample auto-correlated function* (SAC) to characterize traffic stability. Using these features, the system builds classifiers that vote if a user is a crawler. While the heuristic based on HTTP headers has an accuracy of $71.6\,\%$, with the time series analysis it increases to 95%.

CAPTCHAs. CAPTCHAs [59] rely on Turing tests to determine if a user is human. While widespread, their main drawback is that they require user interaction. Moreover, recent progress in image and audio recognition, as well as crowdsourcing services [1], [10] have made it easier to break popular CAPTCHA services such as Google's reCAPTCHA [49], [7].

Behavioral biometrics. Chu *et al.* [11] leverage behavioral biometrics to detect blog spam bots. Their hypothesis is that humans need their mouse to navigate and their keyboards to type. They collect events, such as keystrokes, and train a decision tree to predict if a user is human.

Other detection techniques. The use of IP address lists of cloud providers, proxies and VPNs [63], or the timing of operations to detect browsers in virtual machines [29], can also be used to indicate crawlers.

B. Browser Fingerprinting for Crawler Detection

While the privacy implications of browser fingerprinting have been broadly studied [20], [58], [43], [2], [22], [36], [42], [39], its ability to detect crawlers has been less studied. Recent studies [35], [34] show that websites and bot detection companies heavily rely on the presence of attributes added by instrumentation frameworks and headless browsers, such as `navigator.webdriver`, to detect crawlers. Bursztein *et al.* [9] rely on canvas fingerprinting [39] to create dynamic challenges to detect emulated or spoofed devices used to post fake reviews on the App Store. Contrary to techniques that uniquely identify a device, they verify the devices class by drawing geometric shapes and texts that are expected to render similarly on devices of the same class.

Nikiforakis *et al.* [43], [2] and Vastel *et al.* [57] identify fingerprinting countermeasures, such as user agent spoofers and canvas countermeasures, due to their side effects. In this paper we show that fingerprinters exploit similar techniques, such as non-standard features or inconsistent fingerprints, to detect crawlers.

C. Other Forms of Fingerprinting

While this paper focuses on browser fingerprinting—*i.e.* the application layer—other forms operate at the network and protocol layers. These forms only identify classes of devices or clients, but are not immune to spoofing [50], [24]. TCP fingerprinting relies on the IPv4, IPv6 and TCP headers to identify the OS and software sending a request [4]. The TLS protocol can be used to fingerprint a client [8] due to differences in the sets of cipher suites and elliptic curves in the client's TLS implementation.

III. DETECTING CRAWLER BLOCKING AND FINGERPRINTING WEBSITES

In this section we describe our experimental protocol to classify websites that use browser fingerprinting to detect and block crawlers. This protocol is composed of two main steps:

A. Detecting websites that block crawlers. From Alexa's Top 10K, we identify websites that block crawlers. These websites provide an oracle that we use to evaluate the resilience of browser fingerprinting in Section V;

B. Detecting websites that use fingerprinting. Among websites that block crawlers, we detect those that use fingerprinting to do so (and we break down their techniques in Section IV).

A. Detecting Websites that Block Crawlers

By combining multiple crawlers, we analyze all the websites from Alexa's Top 10K that block crawlers based on the user agent.

Crawler 1: ***Obvious to detect.*** The crawler visits each homepage of Alexa Top 10k websites, and up to 4 random links belonging to the same domain. We only crawl 4 links to isolate fingerprinting from traffic analysis. A page is loaded when there are two active network connections or less for $500\,ms$ (`networkidle2` event of the Puppeteer library). If the page is not loaded after $30\,seconds$, we add it to a failed queue that is retried at the end of the crawl. When a page is loaded, the crawler waits for $3\,seconds$, dumps the HTML, and takes a screenshot. Crawler 1 is based on Chromium headless, instrumented using Puppeteer [27]. We do not modify the user agent, `Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_2) AppleWebKit/537.36 (KHTML, like Gecko)` **`HeadlessChrome`**`/72.0.3582.0 Safari/537.36`, thus clearly exposing it to detection techniques. Although the decision to block might be based on other attributes, it still indicates the way the website reacts to crawlers. Because Headless Chrome is very popular for crawling and has replaced older headless browsers, such as

PhantomJS, we consider it unlikely that websites that block crawlers do not attempt to detect Chrome headless. Moreover, its user agent has been added to popular lists used for crawler detection [38]. We also make the hypothesis that websites that try to block bots using more complex techniques, such as traffic shape analysis, are likely to use crawler blacklists. Indeed, these lists have no false positives and can block crawlers before they load a page.

Crawler 2: ***Checking detection occurrences*****.** On another machine that runs in parallel, with a different IP address, crawler 2, with a user agent modified to a vanilla Chrome browser, visits the homepages of the Alexa Top 10K. This crawl serves to verify the results of crawler 1 and, in particular, the pages reporting errors as websites might fake an error to block the previous crawler. The HTML is also dumped and a screenshot taken.

Labeling screenshots of suspicious websites. To label websites as blocking crawlers or not, we developed a web interface that displays the screenshots for each URL taken by crawlers 1 and 2, side-by-side. Each is assigned 1 out of 3 possible labels:

1) "not blocked": the crawler has not been blocked if the page does not show any signs of blocking, such as an explicit message or a CAPTCHA;
2) "blocked": the crawler has been blocked if the page reports an obvious block, such as a CAPTCHA or a message indicating that we are blocked. If an error from crawler 1 indicates that the page is not available or the website is down, but not from crawler 2, we consider it blocked and not an error;
3) "unknown": corresponds to cases where we cannot assess with certainty the crawler has been blocked. This situation can occur because the page timed-out. In cases where both crawlers 1 and 2 report a `403` error, we manually verify the website. To do so, we visit the website using a computer with a residential IP address that has not been used for crawls. If it still returns a `403` error, we classify the URL as "unknown", as it blocks all users, not only crawlers. Otherwise, the URL is classified as "blocked".

B. Detecting Websites that use Fingerprinting

In the second phase, we focus on the websites we labeled as "blocked". We crawl these websites to classify them as either using fingerprinting for crawler detection or not.

Crawler modifications. We apply two modifications to the crawler's fingerprint to escape detection based on HTTP headers. First, we modify the user agent to look like a user agent from a vanilla Chrome. Second, we add an `accept-language` header field as it is not sent by Headless Chrome [26].

The crawler visits each home page of the websites identified as blocking crawlers and, for each, visits up to 3 randomly selected links on the same domain. We only visit 3 links as the goal is to detect websites using fingerprinting on popular pages that can easily be reached. It does not aim to detect fingerprinting on more sensitive pages, such as login or payment pages. We use the same heuristics to check if a page is loaded. If the page fails, it is added to a queue of URLs that are retried. For each URL, the crawler records the attributes commonly accessed in the browser fingerprinting literature [2], [25], [22]. To record the accesses, we inject a JavaScript snippet to override the default behaviors of getters and functions to store, for each script of the page, when it accesses them. We override the properties of the `navigator` and the `screen` objects; functions related to canvas, audio, and WebGL fingerprinting; and access to attributes used by security fingerprinting scripts, such as `window._phantom` or `navigator.webdriver`, which are known to belong to crawlers. We explain the roles of these attributes in more detail in the next subsection. A complete list of the attributes and functions monitored is available in Appendix A. Finally, we consider a website to use fingerprinting for crawler detection if:

1) at least one script called one or more functions related to canvas, WebGL, audio or WebRTC fingerprinting;
2) the script also tries to access one crawler-related attribute, such as `window._phantom` or `navigator.webdriver`; and
3) the script also retrieves at least 12 fingerprinting attributes.

We adopt this definition as there is no clear agreement on how to characterize fingerprinting, in particular when used for crawler detection. For example, Acar *et al.* [2] consider font enumeration as a good indicator. However, as we show in the next section, font enumeration is not the most discriminant feature for crawler detection. Our definition rather ensures that a script accesses a sufficient number of fingerprinting attributes, in particular attributes considered strong indicators of fingerprinting, such as canvas. As we study crawler detection, we add a constraint to check that the script accesses at least one crawler-related attribute, given that these are widely known and show intent to block crawlers [21].

IV. Analyzing Fingerprinting Scripts

We answer **RQ 1** by showing that fingerprinting is widely used among websites that block crawlers, and **RQ 2** by presenting the techniques implemented by the main fingerprinters present on the Alexa Top 10K.

A. Describing our Experimental Dataset

All crawls were conducted in December 2018.

Sites blocking crawlers. Among the 10,000 websites we crawled, we identified 291 websites that block crawlers (2.91%). The median Alexa rank of websites blocking crawlers is $4,946$, against $5,001$ for websites that do not. Thus, there is no significant difference in the distribution of the rank of websites that block crawlers and websites that do not.

Fingerprinting attributes. For each website that blocks crawlers, we study the number of fingerprinting attributes they access. For a given website, we look at the script that accesses the maximum number of distinct fingerprinting attributes. The median number of distinct fingerprinting attributes accessed is 12, while 10 % of the websites access more than 33. Concerning crawler-specific attributes (e.g., `navigator.webdriver`, `window._phantom`), 51.38 % of the websites do not access any, while 10% access 10. Based on our definition of browser fingerprinting, we found 93

that use fingerprinting for crawler detection, which represents 31.96% of the websites that block crawlers (**RQ 1**).

Diversity of fingerprinting scripts. We group fingerprinting scripts by the combination of attributes they access. In total, we observe 20 distinct groups among websites blocking crawlers. While groups may contain essentially the same script from the same company on different sites, we also observe that some companies are present in different clusters because of multiple versions of their script. We focus on the scripts from 4 fingerprinting companies as they represent more than 90 % of the scripts among the websites that block crawlers. Since they have multiple versions of their script, we chose the script that accesses the greatest distinct number of fingerprinting attributes. We decided not to disclose the names of these companies since it does not contribute to the understanding of fingerprinting and our findings could be used by crawler developers to specifically target some websites.

In the remainder of this section, we report on the techniques used by the 4 main fingerprinting scripts to detect crawlers. These scripts collect fingerprinting attributes and either perform a detection test directly in the browser or transmit the fingerprints to a server to perform the detection test. Table I provides an overview of the attributes and tests. For each attribute and script, there are three possible values:

1) ✓ indicates that the script collects the attribute and tests it in the browser—*i.e.* its value is explicitly verified or we know the attribute is used for the detection because of the evaluation conducted in Section V;
2) ∼ indicates that the script collects the attribute, but no test is run directly in the script. This means the value collected may be used server-side. The empirical evaluation we conduct in Section V help us to understand if some attributes are used server-side;
3) The absence of symbol indicates that the attribute is not collected by the script.

The 4 scripts we analyze are obfuscated and we cannot use variable or function names to infer their purposes. Instead, we use access to open source fingerprinting libraries [55], the state-of-the-art literature, as well as an empirical evaluation we conducted in Section V to explain how the attributes are used.

B. Detecting Crawler-Specific Attributes

The first detection technique in the 4 scripts relies on the presence of attributes injected into the JavaScript execution context or the HTML DOM by headless browsers or instrumenting frameworks. For example, in the case of CHROME or FIREFOX, we can detect an automated browser if the `navigator.webdriver` attribute is set to `true`. The scripts also test for the presence of properties added to the `document` object by SELENIUM, such as: 1) `__fxdriver _unwrapped`, 2) `__selenium _unwrapped`, and 3) `__webdriver_script_fn`. Besides SELENIUM, the scripts also detect headless browsers and automation libraries, such PHANTOMJS by checking for the presence of `_phantom`, `callPhantom` and `phantom` in the `window` object.

While the presence of any of these attributes provides a straightforward heuristic to detect crawlers with certainty, these attributes can be easily removed to escape detection. Thus, we investigate more robust detection techniques, based on fingerprint inconsistencies, to overcome this limitation. We structure the inconsistencies searched by fingerprinters into four categories and we present a fifth of common non-fingerprinting tests found in the scripts: 1) browser and version inconsistencies, 2) OS inconsistencies, 3) screen inconsistencies, 4) overridden functions inconsistencies, and 5) other tests.

TABLE I: Fingerprinting tests and the scripts that use them. A ✓ indicates the attribute is collected and a verification test is run in the script. A ∼ indicates the attribute is collected but no tests are run directly in the script.

		Scripts			
	Name of the test	1	2	3	4
	Crawler-related attributes	✓	✓	✓	✓
Browser	productSub	∼	∼	∼	✓
	eval.toString()	✓			✓
	Error properties	✓			✓
	Browser-specific/prefixed APIs	✓	✓	✓	✓
	Basic features	✓		✓	✓
	Different feature behaviour				✓
	Codecs supported		✓		
	HTTP headers	∼	✓	∼	✓
OS	Touch screen support	∼	∼	∼	✓
	Oscpu and platform	∼	∼	∼	✓
	WebGL vendor and renderer	∼	∼		∼
	List of plugins	∼	∼	∼	✓
	List of fonts		∼	∼	
	Screen dimensions	✓	∼	∼	✓
	Overriden attributes/functions	✓	✓	✓	
Other	Events	∼			∼
	Crawler trap	✓			
	Red pill				✓
	Audio fingerprint	∼			
	Canvas fingerprint	∼	∼	∼	
	WebGL fingerprint	∼	∼		

C. Checking Browser Inconsistencies

The first set of verifications found across the 4 scripts aim at verifying if the user agent has been altered. Before we present the tests, we provide a brief overview of inconsistencies and how they reveal crawlers.

Fingerprint inconsistencies can be defined as combinations of fingerprint attributes that cannot be found in the wild. They have been studied in the literature [43], [57] to reveal fingerprinting countermeasures, such as user agent spoofers or anti-canvas fingerprinting extensions. Fingerprinters also use inconsistencies to detect combinations of attributes that cannot be found for non-automated browsers.

1) Explicit browser consistency tests: One script implements some tests similar to the function `getHasLiedBrowser` proposed by FINGERPRINTJS2 [55]:

productSub. It first extracts the browser from the user agent and verifies if it has a consistent `navigator.productSub` value. While originally it held the build number of the browser, it now always returns `20030107` on Chromium-based or Safari browsers, and `20100101` on Firefox;

eval.toString. Then, it runs `eval.toString().length`, which returns the length of the string representation of the native `eval` function. On Safari and Firefox it is equal to 37, on Internet Explorer it is 39, and on Chromium-based browsers it is 33;

Error properties. It throws an exception and catches it to analyze the properties of the error. While some of the properties of the `Error` objects, such as `message` and `name` are standard across different browsers, others, such as `toSource`, exist only in Firefox. Thus, the script verifies that if the `toSource` property is present in the error, then the browser is Firefox.

2) Feature Detection: We present how different features tested across the 4 fingerprinting scripts can be used to reveal inconsistencies in the nature of the browser and its versions, even when the tests are executed server-side.

Browser-specific APIs. It is possible to test for features specific to certain browsers [40]. All the scripts test for the presence of the `window.chrome` object, a utility for extension developers available in Chromium-based browsers, which can also help to reveal Chrome headless. One script tests for the `pushNotification` function in `window.safari` to verify a Safari browser, the presence of `window.opera` for Opera, for Firefox it verifies if the `InstallTrigger` variable is defined, and for Internet Explorer it checks the value returned by `eval("/*@cc_on!@*/false")`, which relies on conditional compilation, a feature available in old versions of Internet Explorer. Another script tests the presence of features whose names are vendor dependent. For example, it verifies that the function `requestAnimationFrame` is present with `msRequestAnimationFrame` or `webkitRequestAnimationFrame`.

Basic features. Two scripts verify the presence of the `bind` function. While this test does not help in detecting recent headless browsers, it is used to detect PhantomJS [48] as it did not have this function. Another script collects the first 100 properties of the `window` object, returned by `Object.keys`, to verify their consistency with the user agent. Finally, one of the scripts tests a set of 18 basic features, such as creating or removing event listeners using `addEventListener` and `removeEventListener`. It also tests other APIs that have been available in mainstream browsers for a long time, such as `Int8Array` [17], which have been included since Internet Explorer 10, or the `MutationObserver` [15] API, available since Internet Explorer 11. Since, the majority of these features are present in all recent versions of mainstream browsers, they can be used to detect non-standard or headless browsers that do not implement them.

Different feature behaviors. Even when a feature is present, its behavior may vary. For example, Vastel [56] showed that Chrome Headless fails to handle permissions [19] consistently. When requesting permissions using two techniques, Chrome Headless returns conflicting values, as verified by one of the scripts.

TABLE II: Support of audio codecs for the main browsers.

Audio codec	Chrome	Firefox	Safari
ogg vorbis	probably	probably	""
mp3	probably	maybe	maybe
wav	probably	maybe	maybe
m4a	maybe	maybe	maybe
aac	probably	maybe	maybe

Another feature whose behavior depends on the browser is the image error placeholder. When an image cannot be loaded, the browser shows a placeholder whose size depends on the browser. On Chromium-based browsers it is `16x16` pixels and does not depend on the zoom level, while on Safari it is `20x20` pixels and depends on the zoom level. In early versions of Chrome headless, there was no placeholder [56], making them detectable when the placeholders are a size of 0 pixels. One of the scripts detects this by creating an image whose `src` attribute points to a non existing URL.

3) Audio & Video Codecs: One of the scripts tests the presence of different audio and video codecs. To do so, it creates an audio and a video element on which it applies the `canPlayType` method to test the availability of audio and video codecs. The `canPlayType` function returns 3 possible values:

1) `"probably"`, which means that the media type appears to be playable,
2) `"maybe"` indicates that it is not possible to tell if the type can be played without playing it,
3) `""`, an empty string indicating that the type cannot be played.

Table II reports on the audio codecs supported by vanilla browsers. It is based on the dataset from Caniuse [16], [14], [18], [13], as well as data collected on the personal website of one of the authors of this paper. We can observe that some codecs are not supported by all browsers, which means that they can be used to check the browser claimed in the user agent.

4) HTTP headers: Contrary to JavaScript features, which are collected in the browser, HTTP headers are collected on the server side. Thus, we cannot directly observe if fingerprinters collect these headers. Nevertheless, because of side-effects, such as being blocked, we observe that all fingerprinters collect at least the user agent header. Moreover, we also detect that 2 of the fingerprinters test for the presence of the `accept-language` header. Indeed, by default, Chrome headless does not send this header. In the evaluation, we show that its absence enables some of the fingerprinters to block crawlers based on Chrome headless.

D. Checking OS Inconsistencies

Only one script among the four performs an explicit OS verification. Nevertheless, it does not mean that others do not conduct such tests on the server side using attributes collected by the fingerprinting script or using other techniques, such as TCP fingerprinting [62].

Explicit OS consistency tests. The set of tests conducted by the only fingerprinter that verifies the OS in its script is similar to the `getHasLiedOs` function of the library FingerprintJS2 [55]. It extracts the OS claimed in the user agent to use it as a reference and then runs the following set of tests:

1) **Touch screen verification.** It tests if the device supports touch screen by verifying the following properties: the presence of the `ontouchstart` property in the object `window` and `navigator.maxTouchPoints` or `navigator.msMaxTouchPoints` are greater than `0`. If the device claims to have touch support, then it should be running one of the following operating systems: Windows Phone, Android or iOS.
2) **Oscpu and platform.** `Oscpu` is an attribute, only available on Firefox, that returns a string representing the platform on which the browser is executing. The script verifies that the OS claimed in the user agent is consistent with the `navigator.oscpu` attribute. For example, if a platform attribute indicates that the device is running on `arm`, then the OS should be Android or Linux. They also conduct similar tests with the `navigator.platform` attribute.

Only one fingerprinter runs the above set of tests directly in its script. Nevertheless, the other three fingerprinting scripts also collect information about the presence of a touch screen, `navigator.platform` and `navigator.oscpu`. Thus, they may run similar verifications on the server side.

WebGL information. Three of the scripts use the WebGL API to collect information about the vendor and the renderer of the graphic drivers. These values are linked to the OS and can be used to verify OS consistency [57]. For example, a renderer containing `"Adreno"` indicates the presence of an Android device, while a renderer containing `"Iris OpenGL"` reveals the presence of MacOS. One of the scripts also verifies if the renderer is equal to `"Mesa OffScreen"`, which is one of the values returned by the first versions of Headless Chrome [53], [56].

List of plugins. The four scripts collect the list of plugins using the `navigator.plugins` property. While some of the plugins are browser dependent and can be used to verify the claimed browser, they can also be used to verify the OS [57].

List of fonts. Two of the fingerprinting scripts collect a list of fonts using JavaScript font enumeration [43]. While it can be used to increase the uniqueness of the fingerprint [23], it can also be used to reveal the underlying OS [46], [57] since some fonts are only found on specific OSes by default.

E. Checking Screen Inconsistencies

The four scripts collect information related to the screen and window sizes. In particular, they all collect the following attributes: `screen.width/ height`, `screen. availWidth/ Height`, `window. innerWidth/ Height`, `window. outerWidth/ Height` and `window. devicePixelRatio`.

For example, the `screen. width` and `screen. height` represent the width and the height of the web-exposed screen, respectively. The `screen.availWidth` and `screen.availHeight` attributes represent the horizontal and vertical space in pixels available to the window, respectively. Thus, one of the scripts verifies that the available height and width are always less than (in case there is a desktop toolbar) or equal to the height and the width. Another property used to detect some headless browsers is the fact that, by definition, `window.outerHeight/Width` should be greater than `window.innerHeight/Width`. Nevertheless, one should be careful when using this test since it does not hold on iOS devices [30] where the `outerHeight` is always equal to `0`.

Overriden Inconsistencies. Crawler developers may be aware of the detection techniques presented in this section and try to hide such inconsistencies by forging the expected responses—*i.e.*, providing a fingerprint that could come from a vanilla browser, and thus not be detected as a crawler. To do so, one solution is to intercept the outgoing requests containing the fingerprint to modify them on the fly, however, this cannot always be easily done when scripts are carefully obfuscated and randomized. Another solution is to use JavaScript to override the functions and getters used to collect the fingerprint attributes. However, when doing this, the developer should be careful to hide the fact she is overriding native functions and attributes. If not, checking the string representation of the functions will reveal that a native function has been intentionally overridden. While a standard execution of `functionName.toString()` returns a string containing `native code` in the case of a native function, it returns the code of the new function if it has been overridden. Thus, we observe that all the scripts check fingerprinting functions, such as `getImageData` used to obtain a canvas value or the `WebRTC` class constructor, have been overridden.

Detection using side effects. Besides looking at the string representation of native functions and objects, one script goes further by verifying the value returned by a native function. It verifies that the `getImageData` function used to collect the value of a canvas has been overridden by looking at the value of specific pixels.

F. Other Non-fingerprinting Attributes

Events. Crawlers may programmatically generate fake mouse movements to fool behavioral analysis detection systems. To detect such events, two of the fingerprinting scripts check that events originate from human actions. If an event has been generated programmatically, the browser sets its `isTrusted` property to `false`. Nevertheless, this approach does not help in detecting crawlers automated using Selenium or the Chrome DevTools protocol, since the events they generate are considered trusted by the browser.

Crawler trap. One script creates a crawler trap using an invisible link with the `"nofollow"` property and appends a unique random identifier to the URL pointed to by the link. Thus, if a user loads the URL, it can be identified as a crawler that does not respect the `nofollow` policy.

Red pill. One script collects a red pill similar to the one presented by Ho *et al.* [29] to test if a browser is running in a virtual machine or an emulated device. The red pill exploits performance differences caused by caching and virtual hardware.

In this section, we showed that 291 sites from the Alexa Top 10K block crawlers using the user agent. Among these, 93 websites (31.96%) use fingerprinting for crawler detection. They use different techniques that leverage attributes added by automated browsers or fingerprint inconsistencies to detect crawlers.

V. Detecting Crawler Fingerprints

In this section, we first evaluate the effectiveness of browser fingerprinting to detect crawlers. Then, we answer **RQ 3** by studying the resilience of browser fingerprinting against an adversary browser who alters its fingerprint to escape detection.

A. Experimental Protocol

Ground truth challenge. The main challenge to evaluate crawler detection approaches is to obtain ground truth labels to assess the evaluation. The typical approach to obtain labels is to request experts from the field to check raw data, such as fingerprints and HTTP logs, and use their knowledge to label these samples. The main problem of this approach is that labels assigned by the experts are as good as the current knowledge of the experts labeling the data. Similarly to machine learning models that struggle to generalize to new data, these experts may be good at labeling old crawlers they have already encountered, but not at labeling new kinds of crawlers they are unaware of, which may artificially increase or decrease the performance of the approach evaluated. To address this issue, we decide to exercise a family of crawlers on websites that have been identified as blocking crawlers. Thus, no matter how the crawler tries to alter its fingerprint, we can always assert that it is a crawler because it is under our control. Then, in order to measure the effectiveness of fingerprinting for crawler detection, we rely on the fact that the crawled websites have been identified as websites that block crawlers. We consider that, whenever they detect a crawler, they will block it. We use this blocking information as an oracle for the evaluation. A solution to obtain the ground truth would have been to subscribe to the different bot detection services. Nevertheless, besides the significant cost, bot detection companies tend to verify the identity of their customers to ensure it is not used by competitors trying to reverse engineer their solution or by bot creators trying to obtain an oracle to maximize their ad-fraud incomes for example.

1) Crawler Family: In order to evaluate the resilience of fingerprinting, we send 7 different crawlers that incrementally modify their fingerprints to become increasingly more difficult to detect. Table III presents the crawlers and the attributes they modify. The first six crawlers are based on Chrome headless for the following reasons:

1) It has become a popular headless browser for crawling. Since its first release, the once popular PhantomJS stopped being maintained [28];
2) It implements the majority of features present in popular non-headless browsers, making therefore its detection more challenging compare to older headless browsers;
3) Older headless browsers, such as PhantomJS (not maintained since March 2018) and SlimerJS (works only with Firefox version < 59 released in 2017), would have been easily detected because of the lack of modern web features [48].

TABLE III: List of crawlers and altered attributes.

Crawler	Attributes modified
Chrome headless based	
Crawler 1	User agent
Crawler 2	Crawler 1 + `webdriver`
Crawler 3	Crawler 2 + `accept-language`
Crawler 4	Crawler 3 + `window.chrome`
Crawler 5	Crawler 4 + `permissions`
Crawler 6	Crawler 5 + screen resolution + codecs + touch screen
Vanilla Chrome based	
Crawler 7	`webdriver`

The last crawler is based on a vanilla Chrome browser. We use this crawler to better understand why blocking occurs, and to assess that crawlers are blocked because of their fingerprint. Indeed, since this crawler is based on a vanilla Chrome, the only difference in its fingerprint is the `navigator.webdriver` attribute. Once this attribute is removed, it can no longer be detected through fingerprinting.

We restrict the evaluation to 7 different crawlers. Ideally, a perfect protocol would randomly mutate fingerprint attributes to provide a fine-grained understanding. However, this was not feasible in practice, as our evaluation requires residential IP addresses of which we have a limited supply, as well as the exponential complexity resulting from testing all attribute permutations on the set of evaluated websites. While we could have used residential proxy services to acquire more residential IP addresses, this approach still has several drawbacks. Mi *et al.* [37] showed that a majority of the devices proposed by residential proxy services did not give their consent. Moreover, since residential proxy services do not provide mechanisms to ensure the nature of the device that will act as a proxy, there can be inconsistencies between the browser fingerprint of our crawlers and the TCP or TLS fingerprints of the proxy, making it more difficult to understand why a crawler was blocked.

Details of the modified attributes. Crawlers 2 to 6 build on the previous one, adding new modifications each time to increase the difficulty of detection. For example, crawler 4 implements the changes made by crawlers 1, 2 and 3.

1) Crawler 1 is based on Headless Chrome with a modified user agent to look like a vanilla Chrome user agent;
2) In the case of Crawler 2, we delete the `navigator.webdriver` property;
3) By default, Chrome headless does not add an `accept-language` header to its requests. Thus, for Crawler 3, we add this header whose value is set to `"en-US"`;
4) Crawler 4 injects a `chrome` property to the `window` object;
5) For Crawler 5, we override the management of the permissions for the notifications to hide the inconsistency exposed by Headless Chrome [56]. Since we override the behavior of native functions, we also override their `toString` method, as well as

`Function.prototype.toString`—*i.e.*, in order to hide our changes;

6) For Crawler 6, we apply modifications related to the size of the screen, the availability of touch support and the codecs supported by the browser. First, we override the following properties of the `window` object: `innerWidth/Height`, `outerWidth/Height` and `window.screenX/Y`. We also modify properties of the `screen` object: `availWidth/Height` and `width/height`. By default, Chrome headless simulates touch screen support even when Chrome headless is running on a device that does not support it. To emulate a desktop computer without touch support, we override the `document.createEvent` function so that it throws an exception when trying to create a `TouchEvent`. We also override `navigator.maxTouchPoints` to return `0` and we delete the `ontouchstart` property of the `window` object. We also lie about the codecs supported to return the same value as a vanilla Chrome. In order to hide changes made to native functions, we override their `toString`;
7) Contrary to the first six crawlers, Crawler 7 is based on a vanilla Chrome—*i.e.*, non-headless. Thus, we only remove the `webdriver` attribute from the `navigator` object.

2) Evaluation Dataset: We present how we select websites used for the evaluation.

Cross-domain detection. Since we want to evaluate fingerprinting for crawler detection, we try to eliminate other detection factors that could interfere with our evaluation. One such factor is cross-domain detection. This occurs when a company provides a crawler detection service that is present on multiple domains being crawled. In this situation, the company can leverage metrics collected on different domains, such as the number of requests, to classify traffic no matter the website. In order to minimize the risk that cross-domain detection interferes with our evaluation, we need to decrease the number of websites that belong to the same company in the evaluation dataset. Thus, there is a tradeoff between the number of websites in the evaluation dataset and the capacity to eliminate other factors, such as cross-domain detection. While, to our knowledge, no research has been published on cross-domain detection, we encountered this phenomenon during the different crawls we conducted. Moreover, during informal discussions with a crawler detection company, engineers also mentioned this practice.

Selection of websites. We group websites identified as blocking crawlers and using fingerprinting (as defined in Section III) based on the combination of fingerprinting attributes they access. We obtain 20 groups of fingerprinting scripts and, for each of the groups, we randomly select one website. Even though it does not totally eliminate cross-domain detection since, as shown in Section IV, fingerprinters can have different scripts, it still enables to evaluate all the different fingerprinting scripts present in the dataset. Then, we randomly select 20 websites that block crawlers without using fingerprinting to compare fingerprinting-based detection against other approaches.

Crawling protocol. For each of the 7 crawlers, we run 5 crawls on the previously selected websites. Each crawl is run from a machine with a residential or university IP address that has not been used for crawling for at least 2 days to limit the influence of IP reputation. Studying how IP reputation influences detection is left as future work. A crawl consists of the following steps:

a) We randomly shuffle the order of the websites in the evaluation dataset. It enables to minimize and measure the side effects that can occur because of cross-domain detection;
b) For each website, the crawler visits the home page and then visits up to 10 randomly-selected pages from the same domain. As explained later in this section, we crawl only 10 links to ensure that we evaluate the effectiveness of browser fingerprinting detection and not the effectiveness of other state-of-the-art detection approaches;
c) Once a page is loaded, the crawler takes a screenshot and stores the HTML of the page for further analysis;
d) Between two consecutive crawled pages, the crawler waits for 15 seconds plus a random time between 1 and 5 seconds.

3) Crawler behaviors: In the previous subsection, we explain how we select websites in a way that minimizes cross-domain detection. Here, we present how we adapt the behavior of the 7 crawlers so that other approaches, such as rate limiting techniques or behavioral analysis, do not interfere with our evaluation. Thus, the crawlers should not be detected by state-of-the-art techniques presented in Section II that rely on the following features: 1) Number of HTTP requests, 2) Number of bytes requested from the server, 3) Number and percentage of HTML requests, 4) Percentage of PDF requests, 5) Percentage of image requests, 6) Duration of a session, 7) Percentage of `4xx` error requests, 8) ROBOTS.TXT file request, 9) Page popularity index, and 10) Hidden links.

To address points (1) to (6), crawlers request few pages so that it looks like the requests originate from a human. Moreover, we do not block any resources, such as images or PDFs, nor do we ask for these resources in particular. The crawlers visit only up to 10 pages for a given website. Since the attributes used in fingerprinting are constant on short time periods, such as a crawling session, a fingerprinter does not need multiple pages to detect if a fingerprint belongs to a crawler, which means that this criterion should not affect our evaluation. Moreover, the navigation delay between 2 pages is 15 seconds plus a random delay between 1 and 5 seconds. We chose a mean value of 15 seconds since it has been observed that a majority of users do not stay more than 15 seconds on a page on average [31]. We add some randomness so that if a website measures the time between two requested pages, it does not look deterministic. Points (7) and (9) are addressed by only following internal links exposed from the home page or pages directly linked by the home page, which is more likely to point to both popular and existing pages. To address point (8), the crawlers never request the `Robots.txt` file, which means that we do not take into account the policy of the website concerning crawlers. Nevertheless, since we crawl only a few pages, it should have little impact.

B. Experimental Results

1) Presentation of the dataset: In total, we crawl 40 different websites, randomly selected from the list of websites

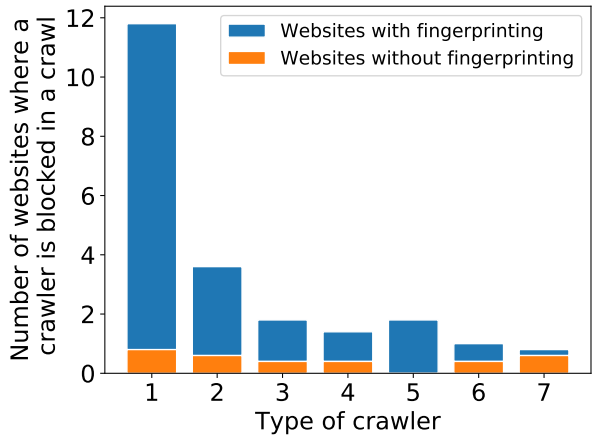

Fig. 2: Crawl efficiency statistics.

blocking crawlers between December 2018 and January 2019. 22 of them use browser fingerprinting and 18 do not use fingerprinting. Initially, we selected two equal sets of 20 websites *using* and *not using* fingerprinting. Nevertheless, we noticed that 2 of the websites had been misclassified. We did not detect fingerprinting on these websites but we observed the side-effects of cross-domains fingerprinters. Since the crawler used for fingerprinting detection had been detected on some other websites, its IP address was blacklisted. Thus, when the crawler visited other websites with the fingerprinter that blocked it earlier, it was blocked at the first request because of its IP address, without having the possibility to load and execute the JavaScript present on the page. In total, we run 35 crawls—*i.e.*, 5 per crawler—each with a residential IP address that has not been used for at least two days for crawling.

2) Blocking results: Figure 2 reports on the results of the crawls for the 7 crawlers. The results have been obtained by labeling the data using the same web interface as the one we used in Section III. For each crawler, we present the average number of times per crawl it is blocked by websites that use fingerprinting and websites that do not use fingerprinting.

Influence of fingerprint modifications. We see that the more changes are applied to the crawler's fingerprint, the less it gets blocked. While Crawler 1 gets blocked 11.8 times on average, the detection falls to 1.0 time for Crawler 6 that applies more extensive modifications to its fingerprint. We also observe an important decrease in the number of times crawlers are blocked between crawlers 1 and 2. It goes from 11.8 for Crawler 1 to 3.6 for Crawler 2. The only difference being the removal of the `webdriver` attribute from the `navigator` object, which means that fingerprinters heavily rely on this attribute to detect crawlers.

Blocking speed. We also analyze the speed at which crawlers get blocked—*i.e.*, after how many pages crawled on a given website a crawler is blocked. Except for Crawler 5 that gets blocked after 3.1 pages crawled on average, crawlers are blocked before crawling 3 pages of a website, on average.

Fingerprinters detect more crawlers. We also observe that, on average, websites using fingerprinting block more crawlers than websites without fingerprinting. For example, on average, 93.2 % (11.0) of websites blocking Crawler 1 use fingerprinting. The only exception is Crawler 7, where 75 % of the time it gets blocked, it is by a website not using fingerprinting. This is the expected result since Crawler 7 is based on a vanilla Chrome, which means that its fingerprint is not different from the one of a standard browser.

Analysis of other detection factors. The fact that Crawler 7 still gets blocked despite the fact it has a normal fingerprint raises the question of other detection factors used in addition to fingerprinting. Even though we take care to adapt the behavior of the crawlers to minimize the chance they get detected by other techniques, we cannot exclude that it occurs. Thus, we verify if crawlers are detected because of their fingerprint or because of other state-of-the-art detection techniques.

First, we investigate if some of the crawlers have been blocked because of cross-domain detection. To do so, we manually label, for each fingerprinting script in the evaluation dataset, the company it belongs to. Whenever we cannot identify the company, we assign a random identifier. We identify 4 fingerprinters present on more than 2 websites in the evaluation dataset and that could use their presence on multiple domains to do cross-domain detection. We focus only on websites that blocked Crawlers 4, 5 and 6. Indeed, only one fingerprinting company succeeds to detect Crawlers 4, 5 and 6. Thus, we argue that Crawlers 1, 2 and 3 detected by websites using fingerprinting, are indeed detected because of their fingerprint. If their detection had relied on other techniques, then some of the Crawlers 4, 5, 6 and 7 would have also been blocked by these websites. Moreover, the analysis of the fingerprinting scripts we conduct in Section IV shows that some of these fingerprinters have the information needed to detect Crawlers 1, 2 and 3, but not to detect more advanced crawlers using fingerprinting.

We analyze in more details the only fingerprinter that detected Crawlers 4, 5 and 6. At each crawl, the order of the websites is randomized. Thus, for each crawler and each crawl, we extract the rank of each of the websites that have a fingerprinting script from this company. Then, we test if the order in which websites from this fingerprinter are crawled impact the chance of a crawler to be detected. Nevertheless, we observe that crawlers get blocked on websites independently of their rank.

Non-stable blocking behavior. We also notice that websites that use the fingerprinting scripts provided by the only fingerprinter that blocked crawlers 4, 5 and 6 do not all behave the same way. Indeed, depending on the website, some of the advanced crawlers have never been blocked. It can occur for several reasons: 1) The websites have different versions of the scripts that collect different attributes; 2) On its website, the fingerprinter proposes different service plans. While some of them are oriented towards blocking crawlers, others only aim at detecting crawlers to improve the quality of the analytics data.

Even on the same website, the blocking behavior is not always stable over time. Indeed, some of the websites do not always block a given crawler. Moreover, some of the websites able to block advanced crawlers do not block crawlers easier to detect. For example, the only website that is able to block both crawlers 5 and 6, only blocked 13 times over the 35 crawls made by all the crawlers. It means that 37.1% of the time, this website did not block crawlers, even though it could have done

so. In particular, this website never blocked Crawlers 1 and 2 even though they are easier to detect than Crawlers 5 and 6.

Undetected crawlers. We also observe that some websites could have detected Crawlers 3 and 4 using the information they collected. Indeed, these websites verify the consistency of the notification permission, which as we show in Section IV, enables to detect crawlers based on Chrome headless. A possible explanation to why the fingerprinter present on these websites was blocking Crawlers 1, 2, but not Crawlers 3 and 4 is because the first two crawlers can be detected solely using information contained in the HTTP headers (lack of `accept-language` header). However, Crawlers 3 and 4 require information collected in the browser, which may be handled differently by the fingerprinter.

In this section, we showed that fingerprinting helps to detect more crawlers than non-fingerprinting techniques. For example, 93.2% (11 websites) of the websites that have detected crawler 1 use fingerprinting. Nevertheless, the important decrease in the average number of times crawlers are blocked between crawlers 1 and 2, from 11.8 websites to 3.6, indicates that websites rely on simple features such as the presence of the `webdriver` attribute to block crawlers. Finally, we show that only 2.5% of the websites detect crawler 6 that applied heavier modifications to its fingerprint to escape the detection, which shows one of the main flaws of fingerprinting for crawler detection: its lack of resilience against adversarial crawlers.

VI. Discussions

A. Limits of Browser Fingerprinting

The analysis of the major fingerprinting scripts shows that it is heavily used to detect older generations of headless browsers or automation frameworks. These browsers and frameworks used to be easily identifiable because of the attributes they injected in the `window` or `document` objects. In addition to these attributes, older headless browsers lacked basic features that were present by default in mainstream browsers, making them easily detectable using feature detection. Since 2017, Chrome headless has proposed a more realistic headless browser that implements most of the features available in a vanilla Chrome. Even though we show that fingerprinters use differences between vanilla Chrome and headless Chrome for detection, it is much harder to find such differences compared to older headless browsers. Thus, since there are fewer differences, it makes it easier for an adversarial crawler developer to escape detection by altering the fingerprint of her crawlers. Indeed, these changes require few lines of code (less than 300 lines in the case of Crawler 6) and can be done directly in JavaScript without the need to modify and compile a whole Chromium browser.

B. Future of fingerprinting

One of the main challenges for fingerprinting-based detection relates to the discovery of new rules. Fingerprinters continuously need to define new rules to detect new browsers or new versions of such browsers, as well as to detect crawlers whose fingerprints have been intentionally modified. Since it is a cumbersome and error-prone task, we argue that there is a need for automation. Schwarz *et al.* [47] proposed an approach to automatically learn the differences between different browsers running on different OSes. Their approach could be applied to headless browsers and extended to take into account more complex differences that require special objects to be instantiated or functions that need to be called sequentially. We also argue there is a need for more complex attributes whose values are harder to spoof, based on APIs such as canvas, WebGL or audio. Since a significant part of crawlers is run from virtual machines in the cloud, we also argue in favor of reliable red pills that can be executed in the browser.

C. Threats to Validity

While the goal of our study is to evaluate the effectiveness of browser fingerprinting for crawler detection, a threat lies in the possibility that we may have missed external techniques, other than browser fingerprinting and the techniques presented in Section II, that could have contributed to the detection of crawlers. A second threat lies in the choice of our oracle—*i.e.*, being blocked by a website when a crawler is detected. While we ensured that all the websites used in the evaluation block crawlers upon detection, it may have been caused by some user agent blacklisting. Thus, we make the hypothesis that, if fingerprinting was also used for crawler detection, then the website would be consistent in its strategy against the crawlers. However, it is possible that a website adopts fingerprinting not against crawlers, but against credit card fraudsters or to label crawlers in its analytics reports, and thus does not focus on blocking crawlers. Finally, a possible threat lies in our experimental framework. We did extensive testing of our code, and we manually verified the data from our experiments. However, as for any experimental infrastructure, there may be bugs. We hope that they only change marginal quantitative results and not the quality of our findings.

VII. Conclusion

Crawler detection has become widespread among popular websites to protect their data. While existing approaches, such as CAPTCHAs or trafic shape analysis, have been shown to be effective, they either require the user to solve a difficult problem, or they require enough data to accurately classify the traffic.

In this paper, we show that, beyond its adoption for tracking, browser fingerprinting is also used as a crawler detection mechanism. We analyze the scripts from the main fingerprinters present in the Alexa Top 10K and show that they exploit the lack of browser features, errors or overridden native functions to detect crawlers. Then, using 7 crawlers that apply different modifications to their fingerprint, we show that websites with fingerprinting are better and faster at detecting crawlers compared to websites that use other state-of-the-art detection techniques. Nevertheless, while 29.5% of the evaluated websites are able to detect our most naive crawler that applies only one change to its fingerprint, this rate decreases to 2.5% for the most advanced crawler that applies more extensive modifications to its fingerprint. We also show that fingerprinting does not help detecting crawlers based on standard browsers since they do not expose inconsistent fingerprints.

APPENDIX

1. Navigator properties.

1) userAgent,
2) platform,
3) plugins,
4) mimeTypes,
5) doNotTrack,
6) languages,
7) productSub,
8) language,
9) vendor,
10) oscpu,
11) hardwareConcurrency,
12) cpuClass,
13) webdriver,
14) chrome.

2. Screen properties.

1) width,
2) height,
3) availWidth,
4) availHeight,
5) availTop,
6) availLeft,
7) colorDepth,
8) pixelDepth.

3. Window properties.

1) ActiveXObject,
2) webdriver,
3) domAutomation,
4) domAutomationController,
5) callPhantom,
6) spawn,
7) emit,
8) Buffer,
9) awesomium,
10) _Selenium_IDE_Recorder,
11) __webdriver_script_fn,
12) _phantom,
13) callSelenium,
14) _selenium.

4. Audio methods.

1) createAnalyser,
2) createOscillator,
3) createGain,
4) createScriptProcessor,
5) createDynamicsCompressor,
6) copyFromChannel,
7) getChannelData,
8) getFloatFrequencyData,
9) getByteFrequencyData,
10) getFloatTimeDomainData,
11) getByteTimeDomainData.

5. WebGL methods.

1) getParameter,
2) getSupportedExtensions,
3) getContextAttributes,
4) getShaderPrecisionFormat,
5) getExtension,
6) readPixels,
7) getUniformLocation,
8) getAttribLocation.

6. Canvas methods.

1) toDataURL,
2) toBlob,
3) getImageData,
4) getLineDash,
5) measureText,
6) isPointInPath.

7. WebRTC methods.

1) createOffer,
2) createAnswer,
3) setLocalDescription,
4) setRemoteDescription.

8. Other methods.

1) Date.getTimezoneOffset,
2) SVGTextContentElement.getComputedTextLength

REFERENCES

[1] 2Captcha. (2018) Online captcha solving and image recognition service. [Online]. Available: https://2captcha.com/

[2] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel, "Fpdetective: dusting the web for fingerprinters," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1129–1140.

[3] F. Alaca and P. C. van Oorschot, "Device fingerprinting for augmenting web authentication: classification and analysis of methods," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 289–301.

[4] B. Anderson and D. McGrew, "Os fingerprinting: New techniques and a study of information gain and obfuscation," in *2017 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2017, pp. 1–9.

[5] A. Balla, A. Stassopoulou, and M. D. Dikaiakos, "Real-time web crawler detection," in *Telecommunications (ICT), 2011 18th International Conference on*. IEEE, 2011, pp. 428–432.

[6] B. Bernard. (2018) Web scraping and crawling are perfectly legal, right? [Online]. Available: https://benbernardblog.com/web-scraping-and-crawling-are-perfectly-legal-right/

[7] K. Bock, D. Patel, G. Hughey, and D. Levin, "uncaptcha: a low-resource defeat of recaptcha's audio challenge," in *Proceedings of the 11th USENIX Conference on Offensive Technologies*. USENIX Association, 2017, pp. 7–7.

[8] L. Brotherson. (2015) Tls fingerprinting. [Online]. Available: https://blog.squarelemon.com/tls-fingerprinting/

[9] E. Bursztein, A. Malyshev, T. Pietraszek, and K. Thomas, "Picasso: Lightweight device class fingerprinting for web clients," in *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 2016, pp. 93–102.

[10] A. CAPTCHA. (2018) Anti captcha: captcha solving service. bypass recaptcha, funcaptcha, image captcha. [Online]. Available: https://anti-captcha.com/mainpage

[11] Z. Chu, S. Gianvecchio, A. Koehl, H. Wang, and S. Jajodia, "Blog or block: Detecting blog bots through behavioral biometrics," *Computer Networks*, vol. 57, no. 3, pp. 634–646, 2013.

[12] S. C. L. D. Commons. (2016) Complaint for violation of the computer fraud and abuse act. [Online]. Available: https://digitalcommons.law.scu.edu/cgi/viewcontent.cgi?article=2261&context=historical

[13] A. Deveria. (2019) Support of advanced audio coding format. [Online]. Available: https://caniuse.com/#feat=aac

[14] ——. (2019) Support of mp3 audio format. [Online]. Available: https://caniuse.com/#feat=mp3

[15] ——. (2019) Support of mutation observers. [Online]. Available: https://caniuse.com/#search=MutationObserver

[16] ——. (2019) Support of ogg vorbis audio format. [Online]. Available: https://caniuse.com/#search=ogg

[17] ——. (2019) Support of typed arrays. [Online]. Available: https://caniuse.com/#search=Int8Array

[18] ——. (2019) Support of waveform audio file format. [Online]. Available: https://caniuse.com/#search=wav

[19] M. W. Docs. (2018) Permissions api. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Permissions_API

[20] P. Eckersley, "How unique is your web browser?" in *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 2010, pp. 1–18.

[21] E.-C. Eelmaa. (2016) Can a website detect when you are using selenium with chromedriver? [Online]. Available: https://stackoverflow.com/questions/33225947/can-a-website-detect-when-you-are-using-selenium-with-chromedriver/41220267#41220267

[22] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1388–1401.

[23] D. Fifield and S. Egelman, "Fingerprinting web users through font metrics," in *International Conference on Financial Cryptography and Data Security*. Springer, 2015, pp. 107–124.

[24] S. Frolov and E. Wustrow, "The use of TLS in censorship circumvention," in *Network and Distributed System Security*. The Internet Society, 2019. [Online]. Available: https://tlsfingerprint.io/static/frolov2019.pdf

[25] A. Gómez-Boix, P. Laperdrix, and B. Baudry, "Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale," in *WWW 2018: The 2018 Web Conference*, 2018.

[26] Google. (2017) Issue 775911 in chromium: missing accept languages in request for headless mode. [Online]. Available: https://groups.google.com/a/chromium.org/forum/#!topic/headless-dev/8YujuBps0oc

[27] ——. (2019) Puppeteer. [Online]. Available: https://pptr.dev/

[28] A. Hidayat. (2019) Phantomjs - scriptable headless browser. [Online]. Available: http://phantomjs.org/

[29] G. Ho, D. Boneh, L. Ballard, and N. Provos, "Tick tock: Building browser red pills from timing side channels." in *WOOT*, 2014.

[30] A. Inc. (2014) ios sdk release notes for ios 8.0 gm. [Online]. Available: https://developer.apple.com/library/archive/releasenotes/General/RN-iOSSDK-8.0/

[31] I. Incapsula, "Bot traffic report 2016," http://time.com/12933/what-you-think-you-know-about-the-web-is-wrong/, March 2014.

[32] ——, "Bot traffic report 2016," https://www.incapsula.com/blog/bot-traffic-report-2016.html, January 2017.

[33] G. Jacob and C. Kruegel, "PUB CRAWL : Protecting Users and Businesses from CRAWLers," *Protecting Users and Businesses from CRAWLers Gregoire*, 2009.

[34] H. Jonker, B. Krumnow, and G. Vlot, "Fingerprint surface-based detection of web bot detectors," in *European Symposium on Research in Computer Security*. Springer, 2019, pp. 586–605.

[35] J. Jueckstock and A. Kapravelos, "Visiblev8: In-browser monitoring of javascript in the wild," in *Proceedings of the Internet Measurement Conference*. ACM, 2019, pp. 393–405.

[36] P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 878–894.

[37] X. Mi, Y. Liu, X. Feng, X. Liao, B. Liu, X. Wang, F. Qian, Z. Li, S. Alrwais, and L. Sun, "Resident evil: Understanding residential ip proxy as a dark service," in *Resident Evil: Understanding Residential IP Proxy as a Dark Service*. IEEE, 2019, p. 0.

[38] M. Monperrus. (2019) Crawler-user-agents. [Online]. Available: https://github.com/monperrus/crawler-user-agents

[39] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in html5," *Proceedings of W2SP*, pp. 1–12, 2012.

[40] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. Wien, "Fast and reliable browser identification with javascript engine fingerprinting," in *Web 2.0 Workshop on Security and Privacy (W2SP)*, vol. 5. Citeseer, 2013.

[41] D. networks, "2018 bad bot report," https://resources.distilnetworks.com/travel/2018-bad-bot-report, January 2018.

[42] N. Nikiforakis, W. Joosen, and B. Livshits, "Privaricator: Deceiving fingerprinters with little white lies," in *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2015, pp. 820–830.

[43] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *Security and privacy (SP), 2013 IEEE symposium on*. IEEE, 2013, pp. 541–555.

[44] D. Preuveneers and W. Joosen, "Smartauth: dynamic context fingerprinting for continuous user authentication," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 2015, pp. 2185–2191.

[45] Quora. Is scraping and crawling to collect data illegal? [Online]. Available: https://www.quora.com/Is-scraping-and-crawling-to-collect-data-illegal

[46] T. Saito, K. Takahashi, K. Yasuda, T. Ishikawa, K. Takasu, T. Yamada, N. Takei, and R. Hosoi, "OS and Application Identification by Installed Fonts," *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pp. 684–689, 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7474155/

[47] M. Schwarz, F. Lackner, and D. Gruss, "Javascript template attacks: Automatically inferring host information for targeted exploits," in *NDSS*, 2019.

[48] S. Shekyan. (2015) Detecting phantomjs based visitors. [Online]. Available: https://blog.shapesecurity.com/2015/01/22/detecting-phantomjs-based-visitors/

[49] S. Sivakorn, J. Polakis, and A. D. Keromytis, "I'm not a human: Breaking the google recaptcha."

[50] M. Smart, G. R. Malan, and F. Jahanian, "Defeating tcp/ip stack fingerprinting." in *Usenix Security Symposium*, 2000.

[51] A. Stassopoulou and M. D. Dikaiakos, "Web robot detection: A probabilistic reasoning approach," *Computer Networks*, vol. 53, no. 3, pp. 265–278, 2009.

[52] D. Stevanovic, A. An, and N. Vlajic, "Feature evaluation for web crawler detection with data mining techniques," *Expert Systems with Applications*, vol. 39, no. 10, pp. 8707–8717, 2012.

[53] C. B. Tracker. (2016) Support webgl in headless. [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=617551

[54] T. Unger, M. Mulazzani, D. Fruhwirt, M. Huber, S. Schrittwieser, and E. Weippl, "Shpf: Enhancing http (s) session security with browser fingerprinting," in *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*. IEEE, 2013, pp. 255–261.

[55] V. Vasilyev. (2019) Modern and flexible browser fingerprinting library. [Online]. Available: https://github.com/Valve/fingerprintjs2

[56] A. Vastel. (2017) Detecting chrome headless. [Online]. Available: https://antoinevastel.com/bot%20detection/2017/08/05/detect-chrome-headless.html

[57] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy, "Fp-scanner: The privacy implications of browser fingerprint inconsistencies," in *Proceedings of the 27th USENIX Security Symposium*, 2018.

[58] ——, "Fp-stalker: Tracking browser fingerprint evolutions," in *IEEE S&P 2018-39th IEEE Symposium on Security and Privacy*. IEEE, 2018, pp. 1–14.

[59] L. Von Ahn, M. Blum, N. J. Hopper, and J. Langford, "Captcha: Using hard ai problems for security," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2003, pp. 294–311.

[60] G. Wang, T. Konolige, C. Wilson, X. Wang, H. Zheng, and B. Y. Zhao, "You are how you click: Clickstream analysis for sybil detection." in *USENIX Security Symposium*, vol. 9, 2013, pp. 1–008.

[61] Wikipedia. (2013) Craigslist inc. v. 3taps inc. [Online]. Available: https://en.wikipedia.org/wiki/Craigslist_Inc._v._3Taps_Inc.

[62] M. Zalewski. (2019) p0f v3. [Online]. Available: http://lcamtuf.coredump.cx/p0f3/

[63] J. Zhang, A. Chivukula, M. Bailey, M. Karir, and M. Liu, "Characterization of blacklists and tainted network traffic," in *International Conference on Passive and Active Network Measurement*. Springer, 2013, pp. 218–228.