HAL
open science

FP-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies

Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, Romain Rouvoy

To cite this version:

Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, Romain Rouvoy. FP-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies. Proceedings of the 27th USENIX Security Symposium, Aug 2018, Baltimore, United States. hal-01820197

HAL Id: hal-01820197

https://inria.hal.science/hal-01820197v1

Submitted on 21 Jun 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

FP-Scanner:
The Privacy Implications of Browser Fingerprint Inconsistencies

Antoine Vastel
Univ. Lille / Inria
antoine.vastel@univ-lille.fr

Pierre Laperdrix
Stony Brook University
plaperdrix@cs.stonybrook.edu

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

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

Abstract

By exploiting the diversity of device and browser configurations, browser fingerprinting established itself as a viable technique to enable stateless user tracking in production. Companies and academic communities have responded with a wide range of countermeasures. However, the way these countermeasures are evaluated does not properly assess their impact on user privacy, in particular regarding the quantity of information they may indirectly leak by revealing their presence.

In this paper, we investigate the current state of the art of browser fingerprinting countermeasures to study the inconsistencies they may introduce in altered fingerprints, and how this may impact user privacy. To do so, we introduce FP-SCANNER as a new test suite that explores browser fingerprint inconsistencies to detect potential alterations, and we show that we are capable of detecting countermeasures from the inconsistencies they introduce. Beyond spotting altered browser fingerprints, we demonstrate that FP-SCANNER can also reveal the original value of altered fingerprint attributes, such as the browser or the operating system. We believe that this result can be exploited by fingerprinters to more accurately target browsers with countermeasures.

1 Introduction

Recent studies have shown that user tracking keeps increasing among popular websites [2, 4, 23], with motivations ranging from targeted advertising to content personalization or security improvements. State-of-the-art tracking techniques assign a *Unique User IDentifier* (UUID), which is stored locally—either as a cookie or some other storage mechanism (*e.g.*, local storage, E-tags). Nonetheless, to protect users, private browsing modes and extensions automatically delete cookies and clear storages at the end of a session, decreasing the efficiency of the standard tracking techniques.

In 2010, Eckerlsey [3] revealed a stateless tracking technique that can complement traditional stateful tracking: *browser fingerprinting*. This technique combines several non-*Personally Identifiable Information* (PII) made available as browser attributes and reveal the nature of the user device. These attributes are disclosed by querying a rich diversity of JavaScript APIs, and by analyzing HTTP headers sent by the browser. By collecting browser fingerprints composed of 8 attributes, he demonstrated that 83.6% of the visitors of the PANOPTICLICK website could be uniquely identified.

Since browser fingerprinting is stateless, it is difficult for end-users to opt-out or block, and raises several privacy concerns, in particular when it comes to undesired advertising and profiling. In response to these concerns, researchers have developed countermeasures to protect against browser fingerprinting [10, 11, 15, 20]. Most of the countermeasures rely on modifying the fingerprint's attributes to hide their true identity. Nonetheless, this strategy tends to generate inconsistent combinations of attributes called *inconsistencies*, which are used by commercial fingerprinters, like AUGUR[1], or open source libraries, such as FINGERPRINTJS2 [21], to detect countermeasures.

In this paper, we extend the work of Nikiforakis *et al.* [16], which focused on revealing inconsistencies to detect *user agent spoofers*, to consider a much wider range of browser fingerprinting countermeasures. To do so, we introduce FP-SCANNER, a fingerprint scanner that explores fingerprint attribute inconsistencies introduced by state-of-the-art countermeasures in order to detect if a given fingerprint is genuine or not. In particular, we show that none of the existing countermeasures succeed in lying consistently without being detected and that it is even possible to recover the ground value of key attributes, such as the OS or the browser. Then, we discuss how using detectable countermeasures may impact user privacy, in particular how fingerprinters can leverage this information to improve their tracking algorithms.

In summary, this paper reports on 5 contributions to better evaluate the privacy impact of browser fingerprinting countermeasures: *1)* we review the state-of-the-art browser fingerprinting countermeasures, *2)* we propose an approach that leverages the notion of consistency to detect if a fingerprint has been altered, *3)* we implement a fingerprinting script and an inconsistency scanner capable of detecting altered fingerprints at runtime, *4)* we run extensive experiments to detect how fingerprinting countermeasures can be detected using our inconsistency scanner, and *5)* we discuss the impact of our findings on user privacy.

The remainder of this paper is organized as follows. Section 2 overviews the state of the art in the domain of browser fingerprinting before exploring existing browser fingerprinting countermeasures. Then, Section 3 introduces a new test suite to detect altered browser fingerprints. Section 4 reports on an empirical evaluation of our contribution and Section 5 discusses the impact on user privacy, as well as the threats to validity. Finally, we conclude and present some perspectives in Section 6.

2 Background & Motivations

Before addressing the consistency properties of fingerprint attributes (cf. Section 2.3), we introduce the principles of browser fingerprint (cf. Section 2.1) and existing countermeasures in this domain (cf. Section 2.2).

2.1 Browser Fingerprinting in a Nutshell

Browser fingerprinting provides the ability to identify a browser instance without requiring a stateful identifier. This means that contrary to classical tracking techniques—such as cookies—it does not store anything on the user device, making it both harder to detect and to protect against. When a user visits a website, the fingerprinter provides a script that the browser executes, which automatically collects and reports a set of attributes related to the browser and system configuration known as a *browser fingerprint*. Most of the attributes composing a fingerprint come from either JavaScript browser APIs—particularly the `navigator` object—or HTTP headers. When considered individually, these attributes do not reveal a lot of information, but their combination has been demonstrated as being mostly unique [3, 12].

Browser Fingerprints Uniqueness and Linkability. Past studies have covered the efficiency of browser fingerprinting as a way to uniquely identify a browser. In 2010, Eckersley [3] collected around half a million fingerprints to study their diversity. He showed that among the fingerprints collected, 83.6 % were unique when only considering JavaScript-related attributes. With the appearance of new JavaScript APIs, Mowery *et al.* [14] showed how the HTML 5 canvas API could be used to generate a 2D image whose exact rendering depends on the device. In 2016, Laperdrix *et al.* [12] studied the diversity of fingerprint attributes, both on desktop and mobile devices, and showed that even if attributes, like the list of plugins or the list of fonts obtained through Flash, exhibit high entropy, new attributes like canvas are also highly discriminating. They also discovered that, even though many mobile devices, such as iPhones, are standardized, other devices disclose a lot of information about their nature through their user agent. More recently, Gómez-Boix *et al.* [8] analyzed the impact of browser fingerprinting at a large scale. Their findings raise some new questions on the effectiveness of fingerprinting as a tracking and identification technique as only 33.6% of more than two million fingerprints they analyzed were unique.

Besides fingerprint uniqueness, which is critical for tracking, stability is also required, as browser fingerprints continuously evolve with browser and system updates. Eckersley [3] was the first to propose a simple heuristic to link evolutions of fingerprints over time. More recently, Vastel *et al.* [22] showed that, using a set of rules combined with machine learning, it was possible to keep track of fingerprint evolutions over long periods of time.

Browser Fingerprinting Adoption. Several studies using Alexa top-ranked websites have shown a steady growth in the adoption of browser fingerprinting techniques [1, 2, 5, 16]. The most recent, conducted by Englehardt *et al.* [5], observed that more than 5 % of the Top 1000 Global Sites listed by Alexa were using canvas fingerprinting techniques.

2.2 Browser Fingerprinting Countermeasures

In response to the privacy issues triggered by browser fingerprint tracking, several countermeasures have been developed. Among these, we distinguish 5 different strategies of browser fingerprinting countermeasures: *script blocking*, *attribute blocking*, *attribute switching* with preexisting values, *attribute blurring* with the introduction of noise, and *reconfiguration* through virtualization.

While script blocking extensions are not specifically designed to counter browser fingerprinting, they may include rules that block some fingerprinting scripts. Tools belonging to this category include GHOSTERY,[2] NOSCRIPT,[3] ADBLOCK,[4] and PRIVACY BADGER.[5]

A strategy specifically designed against browser fingerprinting is to decrease the entropy of a fingerprint

by blocking access to specific attributes. CANVAS BLOCKER[6] is a FIREFOX extension that blocks access to the HTML 5 canvas API. Besides blocking, it also provides another mode, similar to CANVAS DEFENDER,[7] that randomizes the value of a canvas every time it is retrieved. Thus, it can also be classified in the category of countermeasures that act by adding noise to attributes. BRAVE[8] is a CHROMIUM-based browser oriented towards privacy that proposes specific countermeasures against browser fingerprinting, such as blocking audio, canvas, and WebGL fingerprinting.

Another strategy consists in switching the value of different attributes to break the linkability and stability properties required to track fingerprints over time. ULTIMATE USER AGENT[9] is a CHROME extension that spoofs the browser's user agent. It changes the user agent enclosed in the HTTP requests as the original purpose of this extension is to access websites that demand a specific browser. FP-BLOCK [20] is a browser extension that ensures that any embedded party will see a different fingerprint for each site it is embedded in. Thus, the browser fingerprint can no longer be linked to different websites. Contrary to naive techniques that mostly randomize the value of attributes, FP-BLOCK tries to ensure fingerprint consistency. RANDOM AGENT SPOOFER[10] is a FIREFOX extension that protects against fingerprinting by switching between different device profiles composed of several attributes, such as the user agent, the platform, and the screen resolution. Since profiles are extracted from real browsers configurations, all of the attributes of a profile are consistent with each other. Besides spoofing attributes, it also enables blocking advanced fingerprinting techniques, such as canvas, WebGL or WebRTC fingerprinting. Since 2018, FIREFOX integrates an option to protect against fingerprinting. Like TOR, it standardizes and switches values of attributes, such as the user agent, to increase the anonymity set of its users, and also blocks certain APIs, such as the geolocation or the gamepads API, to decrease the entropy of the fingerprints.

Another way to break linkability is to add noise to attributes. This approach is quite similar to attribute switching, but targeted at attributes that are the result of a rendering process, like canvas or audio fingerprints, during which noise can be added. FPGUARD [6] is a combination of a CHROMIUM browser and a browser extension that aims at both detecting and preventing fingerprinting. They combine blocking, switching and noise techniques. For example, they block access to fonts by limiting the number of fonts that can be enumerated in JavaScript. They switch attribute values for the navigator and screen objects, and also add noise to rendered canvas images. FPRANDOM [10] is a modified version of FIREFOX that adds randomness in the computation of the canvas fingerprint, as well as the audio fingerprint. They focus on these attributes because canvas fingerprinting is a strong source of entropy [12], and these two attributes rely on multimedia functions that can be slightly altered without being noticed by the user. FPRANDOM includes two modes, one in which noise is different at every call and a second mode where noise remains constant over a session. The goal of the second mode is to protect against replay attacks, that is, if a fingerprinter runs the same script twice, the result will be the same and the browser will not be found to be exposing an artificial fingerprint.

Finally, BLINK [11] exploits reconfiguration through virtual machines or containers to clone real fingerprints—*i.e.*, in contrary to countermeasures that lie on their identity by simply altering the values of the attributes collected—BLINK generates virtual environments containing different fonts, plugins, browsers in order to break the stability of fingerprints, without introducing inconsistencies.

Table 1: Overview of fingerprinting countermeasures

	BLINK	FIREFOX	BRAVE	*UA spoofers*	FP-BLOCK	RAS	FPGUARD	FPRANDOM	CANVAS DEFENDER
`User Agent`	✓	✓		✓	✓	✓	✓		
`HTTP Headers`	✓	✓			✓	✓			
`Navigator object`	✓	✓			✓	✓	✓	✓	
`Canvas`	✓		✓		✓	✓	✓	✓	✓
`Fonts`	✓				✓		✓		
`WebRTC`		✓	✓			✓			
`Audio`	✓		✓			✓		✓	
`WebGL`	✓	✓	✓		✓	✓			

Table 1 summarizes the fingerprint's attributes commonly collected by fingerprinters [13], and altered by the countermeasures we introduced in this section. For more complex countermeasures that alter a wider range of attributes, we give more details in Table 2. In both tables, the presence of a checkmark indicates that the given countermeasure either blocks or manipulates the value of the attribute.

2.3 Browser Fingerprint Consistency

As described above, most of the browser fingerprinting countermeasures alter the value of several attributes, either by blocking access to their values, by adding noise or by faking them. However, by altering the fingerprint's attributes, countermeasures may generate a combination of values that could not naturally appear in the wild. In such cases, we say that a browser fingerprint is inconsistent, or *altered*. For example, the information contained in the attribute `user agent` (`UA`) reveals information about the user browser and OS. The following `UA`, `Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like`

Table 2: Altered attributes per countermeasure

Attribute	FP-BLOCK	RAS	FIREFOX	BRAVE
Languages HTTP	✓	✓		
Encoding	✓	✓		
Accept		✓		
User agents	✓	✓	✓	
Plugins	✓	✓		✓
MimeTypes	✓			✓
Fonts JS	✓			
Screen	✓	✓		
appName		✓		
Timezone	✓	✓	✓	
Language JS	✓	✓		
Platform	✓	✓	✓	
Oscpu		✓	✓	
hardwareConcurrency			✓	
media devices			✓	✓
Canvas block		✓	✓	✓
Canvas blur	✓			
WebRTC		✓		✓
WebGL	✓	✓		✓
Audio		✓	✓	✓
BuildID		✓		
Battery	✓	✓		✓
Sensors			✓	

`Gecko) Chrome /57.0.2987.110 Safari/537.36`, reveals different information about the device:

- The browser family as well as its version: `Chrome` version `57.0.2987.110`;
- The browser engine used to display web pages: `AppleWebKit version 537.36`;
- The OS family: `Linux`.

The OS and browser family, reflected by the `UA`, are expected to be consistent with attributes, such as `navigator.platform`, which represents the platform on which the browser is running, namely `Linux x86_64` on Linux, `Win32` for Windows and `MacIntel` on macOS. Beyond altering the raw value of fingerprint attributes, another source of inconsistency relates to the manipulation of native JavaScript functions to fool the fingerprinting process. For example, one way to implement canvas poisoners is to override the native function `toDataURL`, used to generate a Base64 string representation of a canvas, which may however be detected by dumping the internal representation of the function.

Privacy Implications. Nikiforakis *et al.* [16] were the first to identify such consistency constraints and to create a test suite to detect inconsistencies introduced by user agent spoofers. They claimed that, due to the presence of inconsistencies, browsers with user agent spoofers become more distinguishable than browsers without. Thus, the presence of a user agent spoofer may be used by browser fingerprinters to improve tracking accuracy.

In this paper, we go beyond the specific case of user agent spoofers and study if we can detect a wider range of state-of-the-art fingerprinting countermeasures. Moreover, we also challenge the claim that being more distinguishable necessarily makes tracking more accurate. This motivation is strengthened by recent findings from inspecting the code of a commercial fingerprinting script used by AUGUR.[1] We discovered that this script computes an attribute called `spoofed`, which is the result of multiple tests to evaluate the consistency between the `user agent`, the `platform`, `navigator.oscpu`, `navigator.productSub`, as well as the value returned by `eval.toString.length` used to detect a browser. Moreover, the code also tests for the presence of touch support on devices that claim to be mobiles. Similar tests are also present in the widely used open source library FINGERPRINTJS2 [21]. While we cannot know the motivations of fingerprinters when it comes to detecting browsers with countermeasures—*i.e.*, this could be used to identify bots, to block fraudulent activities, or to apply additional tracking heuristics—we argue that countermeasures should avoid revealing their presence as this can be used to better target the browser. Thus, we consider it necessary to evaluate the privacy implications of using fingerprinting countermeasures.

3 Investigating Fingerprint Inconsistencies

Based on our study of existing browser fingerprinting countermeasures published in the literature, we organized our test suite to detect fingerprint inconsistencies along 4 distinct components. The list of components is ordered by the increasing complexity required to detect an inconsistency. In particular, the first two components aim at detecting inconsistencies at the OS and browser levels, respectively. The third one focuses on detecting inconsistencies at the device level. Finally, the fourth component aims at revealing canvas poisoning techniques. Each component focuses on detecting specific inconsistencies that could be introduced by a countermeasure. While some of the tests we integrate, such as checking the values of both user agents or browser features, have already been proposed by Nikiforakis *et al.* [16], we also propose new tests to strengthen our capacity to detect inconsistencies. Figure 1 depicts the 4 components of our inconsistency test suite.

3.1 Uncovering OS Inconsistencies

Although checking the browser's identity is straightforward for a browser fingerprinting algorithm, verifying the host OS is more challenging because of the sandbox mechanisms used by the script engines. In this section, we present the heuristics applied to check a fingerprinted OS attribute.

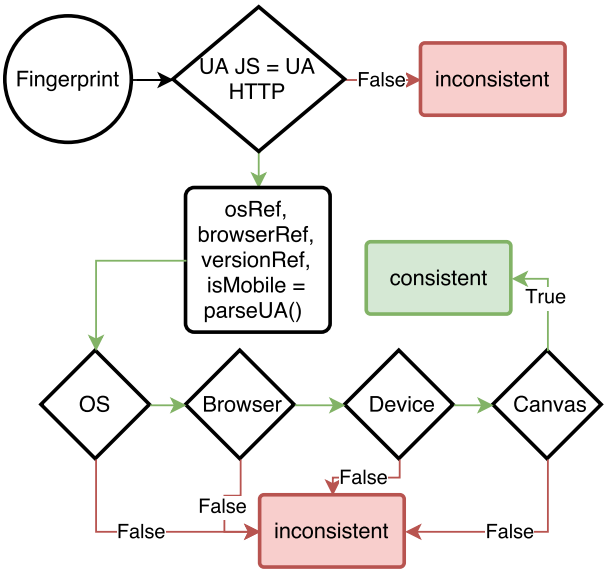


Figure 1: Overview of the inconsistency test suite

User Agent. We start by checking the user agent consistency [16], as it is a key attribute to retrieve the OS and browser of a user. The user agent is available both from the client side, through the navigator object (`navigator.userAgent`), and from the server side, as an HTTP header (`User-Agent`). The first heuristic we apply checks the equality of these two values, as naive browser fingerprinting countermeasures, such as basic user agent spoofers, tend to only alter the HTTP header. The difference between the two user agent attributes reflects a coarse-grained inconsistency that can be due to the OS and/or the browser. While extracting the OS and the browser substrings can help to reveal the source of the inconsistency, the similarity of each substring does not necessarily guarantee the OS and the browser values are true, as both might be spoofed. Therefore, we extract and store the OS, browser and version substrings as internal variables `OSref`, `browserRef`, `browserVersionRef` for further investigation.

Navigator platform. The value of `navigator.platform` reflects the platform on which the browser is running. This attribute is expected to be consistent with the variable `OSref` extracted in the first step [16]. Nevertheless, consistent does not mean equal as, for example, the user agent of a 32-bits Windows will contain the substring `WOW64`, which stands for *Windows on Windows 64-bits*, while the attribute `navigator.platform` will report the value `Win32`. Table 3 therefore maps `OSref` and possible values of `navigator.platform` for the most commonly used OSes.

WebGL. `WebGL` is a JavaScript API that extends the HTML 5 canvas API to render 3D objects from the browser. In particular, we propose a new test that focuses on two WebGL attributes related to the OS:

Table 3: Mapping between common `OS` and `platform` values

OS	Platforms
`Linux`	Linux i686, Linux x86_64
`Windows 10`	Win32, Win64
`iOS`	iPhone, iPad
`Android`	Linux armv71, Linux i686
`macOS`	MacIntel
`FreeBSD`	FreeBSD amd64, FreeBSD i386

Table 4: Mapping between `OS` and substrings in WebGL `renderer`/`vendor` attributes for common OSes

OS	Renderer	Vendor
`Windows`	ANGLE	Microsoft, Google Inc
`macOS`	OpenGL, Iris	Intel, ATI
`Linux`	Mesa, Gallium	Intel, VMWare, X.Org
`Android`	Adreno, Mali, PowerVR	Qualcomm, ARM, Imagination
`Windows Phone`	Qualcomm, Adreno	Microsoft
`iOS`	Apple, PowerVR	Apple, Imagination

`renderer` and `vendor`. The first attribute reports the name of the GPU, for example `ANGLE (VMware SVGA 3D Direct3D11 vs_4_0 ps_4_0)`. Interestingly, the substring `VMware` indicates that the browser is executed in a virtual machine. Also, the `ANGLE` substring stands for *Almost Native Graphics Layer Engine*, which has been designed to bring OpenGL compatibility to Windows devices. The second WebGL attribute (`vendor`) is expected to provide the name of the GPU vendor, whose value actually depends on the OS. On a mobile device, the attribute `vendor` can report the string `Qualcomm`, which corresponds to the vendor of the mobile chip, while values like `Microsoft` are returned for Internet Explorer on Windows, or `Google Inc` for a CHROME browser running on a Windows machine. We therefore summarize the mapping for the attributes `renderer` and `vendor` in Table 4.

Browser plugins. Plugins are external components that add new features to the browser. When querying for the list of plugins via the `navigator.plugins` object, the browser returns an array of plugins containing detailed information, such as their filename and the associated extension, which reveals some indication of the OS. On Windows, plugin file extensions are `.dll`, on macOS they are `.plugin` or `.bundle` and for Linux based OS extensions are `.so`. Thus, we propose a test that ensures that `OSref` is consistent with its associated plugin filename extensions. Moreover, we also consider constraints imposed by some systems, such as mobile browsers that do not support plugins. Thus, reporting plugins on mobile devices is also considered as an inconsistency.

Media queries. Media query is a feature included in CSS 3 that applies different style properties depending on specific conditions. The most common use case is

the implementation of responsive web design, which adjusts the stylesheet depending on the size of the device, so that users have a different interface depending on whether they are using a smartphone or a computer. In this step, we consider a set of media queries provided by the FIREFOX browser to adapt the content depending on the value of desktop themes or Windows OS versions. Indeed, it is possible to detect the Mac graphite theme using `-moz-mac-graphite-theme` media query [19]. It is also possible to test specific themes present on Windows by using `-moz-windows-theme`. However, in the case of Windows, there is a more precise way to detect its presence, and even its version. It is also possible to use the `-moz-os-version` media query to detect if a browser runs on Windows XP, Vista, 7, 8 or 10. Thus, it is possible to detect some Mac users, as well as Windows users, when they are using FIREFOX. Moreover, since these media queries are only available from FIREFOX, if one of the previous media queries is matched, then it likely means that the real browser is FIREFOX.

Fonts. Saito *et al.* [17] demonstrated that fonts may be dependent on the OS. Thus, if a user claims to be on a given OS `A`, but do not list any font linked to this OS `A` and, at the same time, displays many fonts from another OS `B`, we may assume that OS `A` is not its real OS.

This first component in FP-SCANNER aims to check if the OS declared in the user agent is the device's real OS. In the next component, we extend our verification process by checking if the browser and the associated version declared by the user agent have been altered.

3.2 Uncovering Browser Inconsistencies

This component requires the extraction of the variables `browserRef` and `browserVersionRef` from the user agent to further investigate their consistency.

Error In JavaScript, `Error` objects are thrown when a runtime error occurs. There exist 7 different types of errors for client-side exceptions, which depend on the problem that occurred. However, for a given error, such as a stack overflow, not all the browsers will throw the same type of error. In the case of a stack overflow, FIREFOX throws an `InternalError` and CHROME throws a `RangeError`. Besides the type of errors, depending on the browser, error instances may also contain different properties. While two of them—`message` and `name`—are standards, others such as `description`, `lineNumber` or `toSource` are not supported by all browsers. Even for properties such as `message` and `name`, which are implemented in all major browsers, their values may differ for a given error.

For example, executing $null[0]$ on CHROME will generate the following error message *"Cannot read property '0' of null"*, while FIREFOX generates *"null has no properties"*, and SAFARI *"null is not an object (evaluating 'null[0]')"*.

Function's internal representation. It is possible to obtain a string representation of any object or function in JavaScript by using the `toString` method. However, such representations—*e.g.*, `eval.toString()`—may differ depending on the browser, with a length that characterizes it. FIREFOX and SAFARI return the same string, with a length of 37 characters, while on CHROME it has a length of 33 characters, and 39 on INTERNET EXPLORER. Thus, we are able to distinguish most major desktop browsers, except for FIREFOX and SAFARI. Then, we consider the property `navigator.productSub`, which returns the build number of the current browser. On SAFARI, CHROME and OPERA, it always returns the string `20030107` and, combined with `eval.toString().length`, it can therefore be used to distinguish FIREFOX from SAFARI.

Navigator object. `Navigator` is a built-in object that represents the state and the identity of the browser. Since it characterizes the browser, its prototype differs depending not only on the browser's family, but also the browser's version. These differences come from the availability of some browser-specific features, but also from two other reasons:

1. The order of `navigator` is not specified and differs across browsers;
2. For a given feature, different browsers may name it differently. For example, if we consider the feature *getUserMedia*, it is available as `mozGetUserMedia` on FIREFOX and `webkitGetUserMedia` on a Webkit-based browser.

Moreover, as `navigator` properties play an important role in browser fingerprinting, our test suite detects if they have been overridden by looking at their internal string representation. In the case of a genuine fingerprint whose attributes have not been overridden in JavaScript, it should contain the substring `native code`. However, if a property has been overridden, it will return the code of the overridden function.

Browser features. Browsers are complex software that evolve at a fast pace by adding new features, some being specific to a browser. By observing the availability of specific features, it is possible to detect if a browser is the one it claims to be [16]. Since for a given browser, features evolve depending on the version, we can also check if the features available are consistent with `browserVersionRef`. Otherwise, this may indicate that the browser version displayed in the `user agent` has been manipulated.

(a) Canvas fingerprint with no countermeasure

(b) Canvas fingerprint with a countermeasure

Figure 2: (a) a genuine canvas fingerprint without any countermeasures installed in the browser and (b) a canvas fingerprint altered by the Canvas Defender countermeasure that applies a uniform noise to all the pixels in the canvas.

3.3 Uncovering Device Inconsistencies

This section aims at detecting if the device belongs to the class of devices it claims to be—*i.e.*, mobile or computer.

Browser events. Some events are unlikely to happen, such as touch-related events (`touchstart`, `touchmove`) on a desktop computer. On the opposite, mouse-related events (`onclick`, `onmousemove`) may not happen on a smartphone. Therefore, the availability of an event may reveal the real nature of a device.

Browser sensors. Like events, some sensors may have different outputs depending on the nature of devices. For example, the accelerometer, which is generally assumed to only be available on mobile devices, can be retrieved from a browser without requesting any authorization. The value of the acceleration will always slightly deviate from 0 for a real mobile device, even when lying on a table.

3.4 Uncovering Canvas Inconsistencies

Canvas fingerprinting uses the HTML 5 canvas API to draw 2D shapes using JavaScript. This technique, discovered by Mowery *et al.* [14], is used to fingerprint browsers. To do so, one scripts a sequence of instructions to be rendered, such as writing text, drawing shapes or coloring part of the image, and collects the rendered output. Since the rendering of this canvas relies on the combination of different hardware and software layers, it produces small differences from device to device. An example of the rendering obtained on a CHROME browser running on Linux is presented in Figure 2a.

As we mentioned, the rendering of the canvas depends on characteristics of the device, and if an instruction has been added to the script, you can expect to observe its effects in the rendered image. Thus, we consider these scripted instructions as constraints that must be checked in the rendered image. For example, the canvas in Figure 2b has been obtained with the CANVAS DEFENDER extension installed. We observe that contrary to the vanilla canvas that does not use any countermeasure (Figure 2a), the canvas with the countermeasure has a background that is not transparent, which can be seen as a constraint violation. We did not develop a new canvas test, we reused the one adopted by state-of-the-art canvas fingerprinting [12]. From the rendered image, our test suite checks the following properties:

1. *Number of transparent pixels* as the background of our canvas must be transparent, we expect to find a majority of these pixels;
2. *Number of isolated pixels*, which are pixels whose rgba value is different than $(0,0,0,0)$ and are only surrounded by transparent pixels. In the rendered image, we should not find this kind of pixel because shapes or texts drawn are closed;
3. *Number of pixels per color* should be checked against the input canvas rendering script, even if it is not possible to know in advance the exact number of pixels with a given color, it is expected to find colors defined in the canvas script.

We also check if canvas-related functions, such as `toDataUrl`, have been overridden.

4 Empirical Evaluation

This section compares the accuracy of FP-SCANNER, FINGERPRINTJS2 and AUGUR to classify genuine and altered browser fingerprints modified by state-of-the-art fingerprinting countermeasures.

4.1 Implementing FP-Scanner

Instead of directly implementing and executing our test suite within the browser, thus being exposed to countermeasures, we split FP-SCANNER into two parts. The first part is a client-side fingerprinter, which uploads raw browser fingerprints on a remote storage server. For the purpose of our evaluation, this fingerprinter extends state-of-the-art fingerprinters, like FINGERPRINTJS2, with the list of attributes covered by FP-SCANNER (*e.g.*, WebGL fingerprint). Table 5 reports on the list of attributes collected by this fingerprinter. The resulting dataset of labeled browser fingerprints is made available to leverage the reproducibility of our results.[11]

The second part of FP-SCANNER is the server-side implementation, in Python, of the test suite we propose (cf. Section 3). This section reports on the relevant technical issues related to the implementation of the 4 components of our test suite.

4.1.1 Checking OS Inconsistencies

`OSRef` is defined as the OS claimed by the user agent attribute sent by the browser and is extracted using a UA PARSER library.[12] We used the browser fingerprint

Table 5: List of attributes collected by our fingerprinter

Attribute	Description
HTTP headers	List of HTTP headers sent by the browser and their associated value
User agent navigator	Value of navigator.userAgent
Platform	Value of navigator.platform
Plugins	List of plugins (description, filename, name) obtained by navigator.plugins
ProductSub	Value of navigator.productSub
Navigator prototype	String representation of each property and function of the navigator object prototype
Canvas	Base64 representation of the image generated by the canvas fingerprint test
WebGL renderer	WebGLRenderingContext.getParameter("renderer")
WebGL vendor	WebGLRenderingContext.getParameter("vendor")
Browser features	Presence or absence of certain browser features
Media queries	Collect if media queries related to the presence of certain OS match or not using window.matchMedia
Errors type 1	Generate a TypeError and store its properties and their values
Errors type 2	Generate an error by creating a socket not pointing to an URL and store its string representation
Stack overflow	Generate a stack overflow and store the error name and message
Eval toString length	Length of eval.toString().length
mediaDevices	Value of navigator.mediaDevices.enumerateDevices
TouchSupport	Collect the value of navigator.maxTouchPoints, store if we can create a TouchEvent and if window object has the ontouchstart property
Accelerometer	true if the value returned by the accelerometer sensor is different of 0, else false
Screen resolution	Values of screen.width/height, and screen.availWidth/Height
Fonts	Font enumeration using JavaScript [7]
Overwritten properties	Collect string representation of screen.width/height getters, as well as toDataURL and getTimezoneOffset functions

dataset from AMIUNIQUE [12] to analyze if some of the fonts they collected were only available on a given OS. We considered that if a font appeared at least 100 times for a given OS family, then it could be associated to this OS. We chose this relatively conservative value because the AMIUNIQUE database contains many fingerprints that are spoofed, but of which we are unaware of. Thus, by setting a threshold of 100, we may miss some fonts linked to a certain OS, but we limit the number of false positives—*i.e.*, fonts that we would classify as linked to an OS but which should not be linked to it. FP-SCANNER checks if the fonts are consistent with OSRef by counting the number of fonts associated to each OS present in the user font list. If more than $N_f = 1$ fonts are associated to another OS than OSRef, or if no font is associated to OSRef, then FP-SCANNER reports an OS inconsistency. It also tests if moz-mac-graphite-theme and @media(-moz-os-version: $win-version) with $win-version equals to Windows XP, Vista, 7, 8 or 10, are consistent with OSRef.

4.1.2 Checking Browser Inconsistencies

We extract BrowserRef using the same user agent parsing library as for OSRef. With regards to JavaScript errors, we check if the fingerprint has a prototype, an error message, as well as a type consistent with browserRef. Moreover, for each attribute and function of the navigator object, FP-SCANNER also checks if the string representation reveals that it has been overridden.

Testing if the features of the browser are consistent with browserRef is achieved by comparing the features collected using MODERNIZR[13] with the open source data file provided by the website CANIUSE.[14] The file is freely available on Github[15] and represents most of the features present in MODERNIZR as a JSON file. For each of them, it details if they are available on the main browsers, and for which versions. We consider that a feature can be present either if it is present by default or it can be activated. Then, for each MODERNIZR feature we collected in the browser fingerprint, we check if it should be present according to the CANIUSE dataset. If there are more than $N_e = 1$ errors, either features that should be available but are not, or features that should not be available but are, then we consider the browser as inconsistent.

4.1.3 Checking Device Inconsistencies

We verify that, if the device claims to be a mobile, then the accelerometer value is set to true. We apply the same technique for touch-related events. However, we do not check the opposite—*i.e.*, that computers have no touch related events—as some new generations of computers include touch support. Concerning the screen resolution, we first check if the screen height and width have been overridden.

4.1.4 Checking Canvas Poisoning

To detect if a canvas has been altered, we extract the 3 metrics proposed in Section 3. We first count the number of pixels whose rgba value is $(0,0,0,0)$. If the image contains less than $N_{tp} = 4,000$ transparent pixels, or if it is full of transparent pixels, then we consider that the canvas has been poisoned or blocked. Secondly, we count the number of isolated pixels. If the canvas contains more than 10 of them, then we consider it as poisoned. We did not set a lower threshold as we observed that some canvas on macOS and SAFARI included a small number of isolated pixels that are not generated by a countermeasure. Finally, the third metric tests the presence of the orange color $(255,102,0,100)$ by counting the number of pixels having this exact value, and also

Table 6: List of relevant tests per countermeasure.

Test (scope)	*RAS*	*UA spoofers*	*Canvas extensions*	FPRANDOM	BRAVE	FIREFOX
`User Agents` (global)	✓	✓				✓
`Platform` (OS)	✓	✓				✓
`WebGL` (OS)	✓	✓				✓
`Plugins` (OS)	✓	✓				✓
`Media Queries` (`OS, browser`)	✓	✓				✓
`Fonts` (OS)	✓	✓				✓
`Error` (browser)	✓	✓				✓
`Function representation` (browser)	✓		✓			
`Product` (browser)	✓	✓				
`Navigator` (browser)	✓	✓			✓	
`Enumerate devices` (browser)					✓	
`Features` (browser)	✓	✓			✓	✓
`Events` (device)	✓	✓				
`Sensors` (device)	✓	✓				
`toDataURL` (canvas)	✓		✓			
`Pixels` (canvas)	✓		✓	✓	✓	✓

the number of pixels whose color is slightly different—*i.e.*, pixels whose color vector v_c satisfies the following equation $\|(255,102,0,100)-v_c\| < 4$. Our intuition is that canvas poisoners inject a slight noise, thus we should find no or few pixels with the exact value, and many pixels with a slightly different color.

For each test of our suite, FP-SCANNER stores the details of each test so that it is possible to know if it is consistent, and which steps of the analysis failed.

Estimating the parameters Different parameters of FP-SCANNER, such as the number of transparent pixels, may influence the accuracy of FP-SCANNER, resulting in different values of true and false positives. The strategy we use to optimize the value of a given parameter is to run the scanner test that relies on this parameter, and to tune the value of the parameter to minimize the *false positive rate* (FPR)—*i.e.*, the ratio of fingerprints that would be wrongly marked as altered by a countermeasure, but that are genuine. The reason why we do not run all the tests of the scanner to optimize a given parameter is because there may be some redundancy between different tests. Thus, changing a parameter value may not necessarily results in a modification of the detection as a given countermeasure may be detected by multiple tests. Moreover, we ensure that countermeasures are detected for the appropriate symptoms. Indeed, while it is normal for a canvas countermeasure to be detected because some pixels have been modified, we consider it to be a false positive when detected because of a wrong browser feature threshold, as the countermeasure does not act on the browser claimed in the user agent. Table 6 describes, for each countermeasure, the tests that can be used to reveal its presence. If a countermeasure is detected by a test not allowed, then it is considered as a false positive.

Figure 3 shows the detection accuracy and the false positive rate (FPR) for different tests and different values of the parameters to optimize. We define the accuracy as $\frac{\#TP+\#TN}{\#Fingerprints}$ where *true positives* (TP) are the browser fingerprints correctly classified as inconsistent, and *true negatives* (TN) are fingerprints correctly classified as genuine. Table 7 shows, for each parameter, the optimal value we considered for the evaluation. The last column of Table 7 reports on the false positive rate, as well as the accuracy obtained by running only the test that makes use of the parameter to optimize.

Table 7: Optimal values of the different parameters to optimize, as well as the FPR and the accuracy obtained by executing the test with the optimal value.

Attribute	Optimal value	FPR **(accuracy)**
Pixels: N_{tp}	17,200	0 **(0.93)**
Fonts: N_f	2	0 **(0.42)**
Features: N_e	1	0 **(0.51)**

In the case of the number of transparent pixels N_{tp} we observe no differences between 100 and $16,500$ pixels. Between $16,600$ and $18,600$ there is a slight improvement in terms of accuracy caused by a change in the true positive rate. Thus, we chose a value of 17200 transparent pixels since it provides both a false positive rate of 0 while maximizing the accuracy.

Concerning the number of wrong fonts N_f, we obtained an accuracy of 0.646 with a threshold of one font, but this resulted in a false positive rate of 0.197. Thus, we chose a value of $N_f = 2$ fonts, which makes the accuracy of the test decrease to 0.42 but provides a false positive rate of 0.

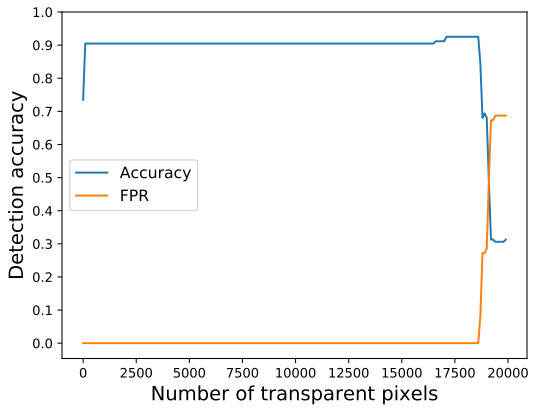
Finally, concerning the number of browser features N_e, increasing the threshold resulted in a decrease of the accuracy, and an increase of the false negative rate. Nevertheless, only the false negative and true positve rates are impacted, not the false positive rate that remains constant for the different values of N_e. Thus, we chose a value of $N_e = 1$

Even if the detection accuracy of the tests may seem low—0.42 for the fonts and 0.51 for the browser features—these are only two tests among multiple tests, such as the `media queries`, `WebGL` or `toDataURL` that can also be used to verify the authenticity of the information provided in the user agent or in the canvas.

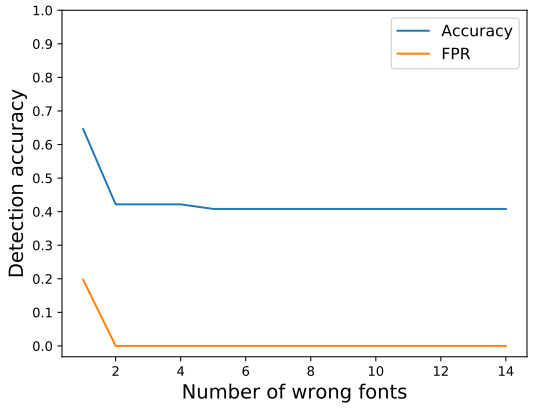
4.2 Evaluating FP-Scanner

4.2.1 Building a Browser Fingerprints Dataset

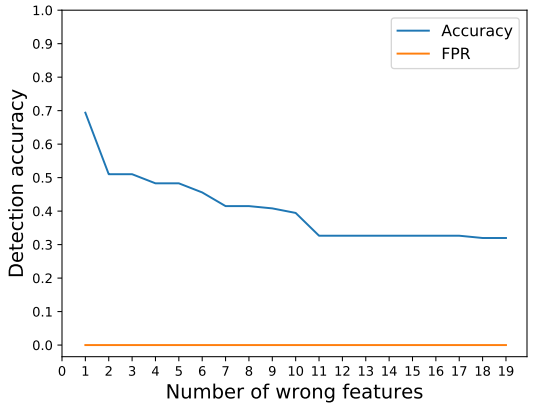
To collect a relevant dataset of browser fingerprints, we created a webpage that includes the browser fingerprinter we designed. Besides collecting fingerprints, we also collect the system ground truth—*i.e.*, the real os, browser family and version, as well as the list of countermeasures



(a) Detection accuracy and false positive rate using the transparent pixels test for different values of N_{tp} (Number of transparent pixels)



(b) Detection accuracy and false positive rate using the fonts test for different values of N_f (Number of fonts associated with the wrong OS)



(c) Detection accuracy and false positive rate of the browser feature test for different values of N_e (Number of wrong features)

Figure 3: Accuracy of the different detection tests for different parameter values

Table 8: Comparison of accuracies per countermeasures

Countermeasure	Number of fingerprints	Accuracy FP Scanner	Accuracy FP-JS2 / Augur
RANDOM AGENT SPOOFER (RAS)	69	**1.0**	0.55
User agent spoofers (UAs)	22	**1.0**	0.86
CANVAS DEFENDER	26	**1.0**	0.0
FIREFOX protection	6	**1.0**	0.0
CANVAS FP BLOCK	3	**1.0**	0.0
FPRANDOM	7	**1.0**	0.0
BRAVE	4	**1.0**	0.0
No countermeasure	10	**1.0**	1.0

installed. In the scope of our experiment, we consider countermeasures listed in Table 8, as they are representative of the diversity of strategies we reported in Section 2. Although other academic countermeasures have been published [6, 11, 15, 20], it was not possible to consider them due to the unavailability of their code or because they could not be run anymore. Moreover, we still consider RANDOM AGENT SPOOFER even though it is not available as a web extension—*i.e.*, for FIREFOX versions > 57—since it modifies many attributes commonly considered by browser fingerprinting countermeasures.

We built this browser fingerprints dataset by accessing this webpage from different browsers, virtual machines and smartphones, with and without any countermeasure installed. The resulting dataset is composed of browser fingerprints, randomly challenged by 7 different countermeasures. Table 8 reports on the number of browser fingerprints per countermeasure. The number of browser fingerprints per countermeasure is different since some countermeasures are deterministic in the way they operate. For example, CANVAS DEFENDER always adds a uniform noise on all the pixels of a canvas. On the opposite, some countermeasures, such as RANDOM AGENT SPOOFER, add more randomness due to the usage of real profiles, which requires more tests.

4.2.2 Measuring the Accuracy of FP-Scanner

We evaluate the effectiveness of FP-SCANNER, FINGERPRINTJS2 and AUGUR to correctly classify a browser fingerprint as *genuine* or *altered*. Our evaluation metric is the accuracy, as defined in Section 4.1. On the globality of the dataset, FP-SCANNER reaches an accuracy 1.0 against 0.45 for FINGERPRINTJS2 and AUGUR, which perform equally on this dataset. When inspecting the AUGUR and FINGERPRINTJS2 scripts, and despite Augur's obfuscation, we observe that they seem to perform the same tests to detect inconsistencies. As the number of fingerprints per countermeasure is unbalanced, Table 8 compares the accuracy achieved per countermeasure.

We observe that FP-SCANNER outperforms FINGERPRINTJS2 to classify a browser fingerprint as *genuine* or

altered. In particular, FP-SCANNER detects the presence of canvas countermeasures while FINGERPRINTJS2 and `Augur` spotted none of them.

4.2.3 Analyzing the Detected Countermeasures

For each browser fingerprint, FP-SCANNER outputs the result of each test and the value that made the test fail. Thus, it enables us to extract some kinds of signatures for different countermeasures. In this section, we execute FP-SCANNER in depth mode—*i.e.*, for each fingerprint, FP-SCANNER executes all of the steps, even if an inconsistency is detected. For each countermeasure considered in the experiment, we report on the steps that revealed their presence.

User Agent Spoofers are easily detected as they only operate on the user agent. Even when both values of user agent are changed, they are detected by simple consistency checks, such as platform for the OS, or function's internal representation test for the browser.

Brave is detected because of the side effects it introduces, such as blocking canvas fingerprinting. FP-SCANNER distinguishes BRAVE from a vanilla Chromium browser by detecting it overrides `navigator.plugins` and `navigator.mimeTypes` getters. Thus, when FP-SCANNER analyzes BRAVE's navigator prototype to check if any properties have been overridden, it observes the following output for plugins and mimeTypes getters string representation: `() => { return handler }`. Moreover, BRAVE also overrides `navigator.mediaDevices.enumerateDevices` to block devices enumeration, which can also be detected by FP-SCANNER as it returns a `Proxy` object instead of an object representing the devices.

Random Agent Spoofer (RAS) By using a system of profiles, RAS aims at introducing fewer inconsistencies than purely random values. Indeed, RAS passes simple checks, such as having identical user agents or having a user agent consistent with `navigator.platform`. Nevertheless, FP-SCANNER still detects inconsistencies as RAS only ensures consistency between the attributes contained in the profile. First, since RAS is a FIREFOX extension, it is vulnerable to the media query technique. Indeed, if the user is on a Windows device, or if the profile selected claims to be on Windows, then the OS inconsistency is directly detected. In the case where it is not enough to detect its presence, plugins or fonts linked to the OS enables us to detect it. Browser inconsistencies are also easily detected, either using function's internal representation test or errors attributes. When only the browser version was altered, FP-SCANNER detects it by using the combination of MODERNIZR and CANIUSE features.

RAS overrides most of the navigator attributes from the FIREFOX configuration file. However, the `navigator.vendor` attribute is overridden in JavaScript, which makes it detectable. FP-SCANNER also detects devices which claimed to be mobile devices, but whose accelerometer value was undefined.

Firefox fingerprinting protection standardizes the user agent when the protection is activated and replaces it with `Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:52.0) Gecko/20100101 Firefox/52.0`, thus lying about the browser version and the operating system for users not on Windows 7 (Windows NT 6.1). While OS-related attributes, such as `navigator.platform` are updated, other attributes, such as `webgl vendor` and `renderer` are not consistent with the OS. For privacy reasons, FIREFOX disabled OS-related media queries presented earlier in this paper for its versions > 57, whether or not the fingerprinting protection is activated. Nevertheless, when the fingerprinting protection is activated, FIREFOX pretends to be version 52 running on Windows 7. Thus, it should match the media query `-moz-os-version` for Windows 7, which is not the case. Additionally, when the browser was not running on Windows, the list of installed fonts was not consistent with the OS claimed.

Canvas poisoners including CANVAS DEFENDER, CANVAS FP BLOCK and FPRANDOM were all detected by FP-SCANNER. For the first two, as they are browser extensions that override canvas-related functions using JavaScript, we always detect that the function `toDataURL` has been altered. For all of them, we detect that the canvas pixel constraints were not enforced from our canvas definition. Indeed, we did not find enough occurrences of the color $(255, 102, 0, 100)$, but we found pixels with a slightly different color. Moreover, in case of the browser extensions, we also detected an inconsistent number of transparent pixels as they apply noise to all the canvas pixels.

Table 9 summarizes, for each countermeasure, the steps of our test suite that detected inconsistencies. In particular, one can observe that FP-SCANNER leverages the work of Nikiforakis *et al.* [16] by succeeding to detect a wider spectrum of fingerprinting countermeasures that were previously escaped by their test suite (*e.g.*, canvas extensions, FPRANDOM [10] and BRAVE). We also observe that the tests to reveal the presence of countermeasures are consistent with the tests presented in Table 6.

4.2.4 Recovering the Ground Values

Beyond uncovering inconsistencies, we enhanced FP-SCANNER with the capability to restore the ground value of key attributes like `OS`, `browser family` and `browser`

Table 9: FP-SCANNER steps failed by countermeasures

Test (scope)	*RAS*	*UA spoofers*	*Canvas extensions*	FPRANDOM	BRAVE	FIREFOX
`User Agents` (global)						
`Platform` (OS)		✓				
`WebGL` (OS)	✓	✓				✓
`Plugins` (OS)	✓	✓				
`Media Queries` (`OS, browser`)	✓	✓				✓
`Fonts` (OS)	✓					✓
`Error` (browser)	✓	✓				
`Function representation` (browser)	✓					
`Product` (browser)	✓	✓				
`Navigator` (browser)	✓				✓	
`Enumerate devices` (browser)					✓	
`Features` (browser)	✓	✓				
`Events` (device)	✓	✓				
`Sensors` (device)	✓	✓				
`toDataURL` (canvas)	✓		✓			
`Pixels` (canvas)			✓	✓	✓	✓

`version`. To recover these attributes, we rely on the hypothesis that some attributes are harder to spoof, and hence more likely to reflect the true nature of the device. When FP-SCANNER does not detect any inconsistency in the browser fingerprint, then the algorithm simply returns the values obtained from the user agent. Otherwise, it uses the same tests used to spot inconsistencies, but to restore the ground values.

OS value To recover the real OS, we combine multiple sources of information, including plugins extensions, WebGL renderer, media queries, and fonts linked to OS. For each step, we obtain a possible OS. Finally, we select the OS that has been predicted by the majority of the steps.

Browser family Concerning the browser family, we rely on function's internal representation (`eval.toString().length`) that we combine with the value of `productSub`. Since these two attributes are discriminative enough to distinguish most of the major browsers, we do not make more tests.

Browser version To infer the browser version, we test the presence or absence of each MODERNIZR feature for the recovered browser family. Then, for each browser version, we count the number of detected features. Finally, we keep a list of versions with the maximum number of features in common.

Evaluation We applied this recovering algorithm to fingerprints altered only by countermeasures that change the OS or the browser—*i.e.*, RAS, *User agent spoofers* and FIREFOX fingerprinting protection. FP-SCANNER was able to correctly recover the browser ground value for 100 % of the devices. Regarding the OS, FP-SCANNER was always capable of predicting the OS family—*i.e.*, Linux, MacOS, Windows—but often failed to recover the correct version of Windows, as the technique we use to detect the version of Windows relies on Mozilla media queries, which stopped working after version 58, as already mentioned. Finally, FP-SCANNER failed to faithfully recover the browser version. Given the lack of discriminative features in MODERNIZR, FP-SCANNER can only recover a range of candidate versions. Nevertheless, this could be addressed by applying natural language processing on browser release notes in order to learn the discriminative features introduced for each version.

4.3 Benchmarking FP-Scanner

This part evaluates the overhead introduced by FP-SCANNER to scan a browser fingerprint. The benchmark we report has been executed on a laptop having an Intel Core i7 and 8 GB of RAM.

Performance of FP-Scanner We compare the performance of FP-Scanner with FINGERPRINTJS2 in term of processing time to detect inconsistencies. First, we automate CHROME HEADLESS version 64 using PUPETEER and we run 100 executions of FINGERPRINTJS2. In case of FINGERPRINTJS2, the reported time is the sum of the execution time of each function used to detect inconsistencies—*i.e.*, `getHasLiedLanguages`, `getHasLiedResolution`, `getHasLiedOs` and `getHasLiedBrowser`. Then, we execute different versions of FP-Scanner on our dataset. Input datasets, such as the CANIUSE features file, are only loaded once, when FP-SCANNER is initialized. We start measuring the execution time after this initialization step as it is only done once. Depending on the tested countermeasure, FP-SCANNER may execute more or less tests to scan a browser fingerprint. Indeed, against a simple user agent spoofer, the inconsistency might be quickly detected by checking the two user agents, while it may require to analyze the canvas pixels for more advanced countermeasures, like FPRANDOM. Thus, in Figure 4, we report on 4 boxplots representing the processing time for the following situations:

1. FINGERPRINTJS2 inconsistency tests,
2. The scanner stops upon detecting one inconsistency (FP-SCANNER (default) mode),
3. All inconsistency tests are executed (FP-SCANNER (depth) mode),
4. Only the test that manipulates the canvas (`pixels` is executed (FP-SCANNER (canvas only) mode).

One can observe that, when all the tests are executed (3)—which corresponds to genuine fingerprints—90 % of the fingerprints are processed in less than 513 *ms*. However, we observe a huge speedup when stopping the processing upon the first occurrence of an inconsistency (2). Indeed, while 83 % of the fingerprints are processed in less than 0.21 *ms*, the remaining 17 % need more than

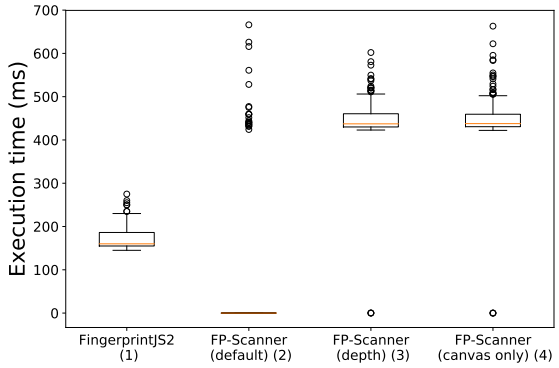


Figure 4: FP-SCANNER execution times

440 *ms*. This is caused by the fact that most of the fingerprints we tested had installed countermeasures that could be detected using straightforward tests, such as media queries or testing for overridden functions, whereas the other fingerprints having either no countermeasures or FPRANDOM (17 fingerprints), require to run all the tests. This observation is confirmed by the fourth boxplot, which report on the performance of the pixel analysis step and imposes additional processing time to analyze all the canvas pixels. We recall that the pixel analysis step is required only to detect FPRANDOM since even other canvas countermeasures can be detected by looking at the string representation of `toDataURL`. Thus, when disabling the pixel analysis test, FP-SCANNER outperforms FINGERPRINTJS2 with a better accuracy (> 0.92) and a faster execution (90th percentile of 220 *ms*).

Based on this evaluation, we can conclude that adopting an inconsistency test suite like FP-SCANNER in production is a viable solution to detect users with countermeasures.

5 Discussion

In this paper, we demonstrated that state-of-the-art fingerprinting countermeasures could be detected by scanning for inconsistencies they introduce in browser fingerprints. We first discuss the privacy implications of such a detection mechanism and then explain how these techniques could be used to detect browser extensions in general.

5.1 Privacy Implications

Discrimination. Being detected with a countermeasure could lead to discrimination. For example, Hannak *et al.* [9] demonstrated that some websites adjust prices depending on the user agent. Moreover, many websites refuse to serve browsers with ad blockers or users of the TOR browser and network. We can imagine users being delivered altered content or being denied access if they do not share their true browser fingerprint. Similarly to ad blocker extensions, discrimination may also happen with a countermeasure intended to block fingerprinting scripts.

Trackability. Detecting countermeasures can, in some cases, be used to improve tracking. Nikiforakis *et al.* [16] talk about the counterproductiveness of using user agent spoofers because they make browsers more identifiable. We extend this line of thought to more generally argue that being detected with a fingerprinting countermeasure can make browsers more trackable, albeit this is not always the case. We assert that the ease of tracking depends on different factors, such as being able to identify the countermeasure, the number of users of the countermeasure, the ability to recover the real fingerprint values, and the volume of information leaked by the countermeasure. To support this claim, we present the countermeasures we studied in this paper.

Anonymity Set. In the case of countermeasures with large user bases, like FIREFOX with fingerprinting protection or BRAVE, although their presence can be detected, these countermeasures tend to increase the anonymity set of their users by blocking different attributes, and, in the case of FIREFOX, by sharing the same user agent, platform, and timezone. Since they are used by millions of users at the time we wrote this paper, the information obtained by knowing that someone uses them does not compensate the loss in entropy from the removal of fingerprinting attributes. On the opposite end, for countermeasures with small user bases, such as CANVAS DEFENDER (21*k* downloads on CHROME, 5*k* on FIREFOX) or RAS (160*k* downloads on FIREFOX), it is unlikely that the anonymity gained by the countermeasures compensate the information obtained by knowing that someone uses them.

Increasing targetability. In the case of RAS, we show that it is possible to detect its presence and recover the original browser and OS family. Also, since the canvas attribute has been shown to have high entropy, and that RAS does not randomize it nor block it by default, the combination of few attributes of a fingerprint may be enough to identify a RAS user. Thus, under the hypothesis that no, or few, RAS users have the same canvas, many of them could be identified by looking at the following subset of attributes: `being a RAS user`, `predicted browser`, `predicted OS`, and `canvas`.

Blurring Noise. In the case of CANVAS DEFENDER, we show that even though they claim to have a safer solution than other canvas countermeasure extensions, the way they operate makes it easier for a fingerprinter to track their users. Indeed, CANVAS DEFENDER applies

a uniform noise vector on all pixels of a canvas. This vector is composed of 4 random numbers between -10 and 30 corresponding to the red, green, blue and alpha (rgba) components of a color. With a small user base, it is unlikely that two or more users share both the same noise and the same original canvas. In particular, the formula hereafter represents the probability that two or more users of CANVAS DEFENDER among k share the same noise vector, which is similar to the birthday paradox: $1 - \prod_{i=1}^{k}(1 - \frac{1}{40^4 - i})$. Thus, if we consider that the $21k$ Chrome users are still active, there is a probability of 0.0082 that at least two users share the same noise vector. Moreover, by default CANVAS DEFENDER does not change the noise vector. It requires the user to trigger it, which means that if a user does not change the default settings or does not click on the button to update the noise, she may keep the same noise vector for a long period. Thus, when detecting that a browser has CANVAS DEFENDER installed, which can be easily detected as the string representation of the `toDataURL` function leaks its code, if the fingerprinting algorithm encounters different fingerprints with the same canvas value, it can conclude that they originate from the same browser with high confidence. In particular, we discovered that CANVAS DEFENDER injects a script element in the DOM (cf. Listing 1). This script contains a function to override canvas-related functions and takes the noise vector as a parameter, which is not updated by default and has a high probability to be unique among CANVAS DEFENDER users. By using the JavaScript Mutation observer API[16] and a regular expression (cf. Listing 2), it is possible to extract the noise vector associated to the browser, which can then be used as an additional fingerprinting attribute.

```
function overrideMethods(docId, data) {
  const s = document.createElement('script')
  s.id = getRandomString();
  s.type = "text/javascript";
  const code = document.createTextNode('try
      {('+overrideDefaultMethods+')(' +data.
      r+ ','+data.g + ',' + data.b + ','+
      data.a+',"'+s.id +'","'+
      storedObjectPrefix+'");}catch(e){
      console.error(e);}');
  s.appendChild(code);
  var node = document.documentElement;
  node.insertBefore(s, node.firstChild);
  node[docId] = getRandomString(); }
```

Listing 1: Script injected by CANVAS DEFENDER to override canvas-related function

```
var o = new MutationObserver((ms) => {
  ms.forEach((m) => {
  var script = "overrideDefaultMethods";
  if (m.addedNodes[0].text.indexOf(script) >
       -1) {
    var noise = m.addedNodes[0].text.match
        (/\d{1,2},\d{1,2},\d{1,2},\d{1,2}/)
        [0].split(",");
} }); });
o.observe(document.documentElement, {
    childList:true, subtree:true});
```

Listing 2: Script to extract the noise vector injected by CANVAS DEFENDER

Protection Level. While it may seem more tempting to install an aggressive fingerprinting countermeasure—*i.e.*, a countermeasure, like RAS, that blocks or modifies a wide range of attributes used in fingerprinting—we believe it may be wiser to use a countermeasure with a large user base even though it does not modify many fingerprinting attributes. Moreover, in the case of widely-used open source projects, this may lead to a code base being audited more regularly than less adopted proprietary extensions. We also argue that all the users of a given countermeasure should adopt the same defense strategy. Indeed, if a countermeasure can be configured, it may be possible to infer the settings chosen by a user by detecting side effects, which may be used to target a subset of users that have a less common combination of settings. Finally, we recommend a defense strategy that either consists in blocking the access to an attribute or unifying the value returned for all the users, rather than a strategy that randomizes the value returned based on the original value. Concretely, if the value results from a randomization process based the original value, as does CANVAS DEFENDER, it may be possible to infer information on the original value.

5.2 Perspectives

In this article, we focused on evaluating the effectiveness of browser fingerprinting countermeasures. We showed that these countermeasures can be detected because of their side-effects, which may then be used to target some of their users more easily. We think that the same techniques could be applied, in general, to any browser extension. Starov et al. [18] showed that browser extensions could be detected because of the way they interact with the DOM. Similar techniques that we used to detect and characterize fingerprinting countermeasures could also be used for browser extension detection. Moreover, if an extension has different settings resulting in different fingerprintable side effects, we argue that these side effects could be used to characterize the combination of settings used by a user, which may make the user more trackable.

5.3 Threats to Validity

A possible threat lies in our experimental framework. We did extensive testing of FP-SCANNER to ensure that

browser fingerprints were appropriately detected as altered. Table 9 shows that no countermeasure failed the steps unrelated to its defense strategy. However, as for any experimental infrastructure, there might be bugs. We hope that they only change marginal quantitative results and not the quality of our findings. However, we make the dataset, as well as the algorithm, publicly available online[11], making it possible to replicate the experiment.

We use a ruleset to detect inconsistencies even though it may be time-consuming to maintain an up-to-date set of rules that minimize the number of false positives while ensuring it keeps detecting new countermeasures. Moreover, in this paper, we focused on browser fingerprinting to detect inconsistencies. Nonetheless, we are aware of other techniques, such as TCP fingerprinting[17], that are complementary to our approach.

FP-SCANNER aims to be general in its approach to detect countermeasures. Nevertheless, it is possible to develop code to target specific countermeasures as we showed in the case of CANVAS DEFENDER. Thus, we consider our study as a lower bound on the vulnerability of current browser fingerprinting countermeasures.

6 Conclusion

In this paper, we identified a set of attributes that is explored by FP-SCANNER to detect inconsistencies and to classify browser fingerprints into 2 categories: *genuine* fingerprints and *altered* fingerprints by a countermeasure. Thus, instead of taking the value of a fingerprint for granted, fingerprinters could check whether attributes of a fingerprint have been modified to escape tracking algorithms, and apply different heuristics accordingly.

To support this study, we collected browser fingerprints extracted from browsers using state-of-the-art fingerprinting countermeasures and we showed that FP-SCANNER was capable of accurately distinguishing genuine from altered fingerprints. We measured the overhead imposed by FP-SCANNER and we observed that both the fingerprinter and the test suite were impose a marginal overhead on a standard laptop, making our approach feasible for use by fingerprinters in production. Finally, we discussed how the possibility of detecting fingerprinting countermeasures, as well as being capable of predicting the ground value of the browser and the OS family, may impact user privacy. We argued that being detected with a fingerprinting countermeasure does not necessarily imply being tracked more easily. We took as an example the different countermeasures analyzed in this paper to explain that tracking vulnerability depends on the capability of identifying the countermeasure used, the number of users having the countermeasure, the capacity to recover the original fingerprint values, and the information leaked by the countermeasure. Although FP-SCANNER is general in its approach to detect the presence of countermeasures, using CANVAS DEFENDER as an example, we show it is possible to develop countermeasure-specific code to extract more detailed information.

References

[1] ACAR, G., EUBANK, C., ENGLEHARDT, S., JUAREZ, M., NARAYANAN, A., AND DIAZ, C. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 674–689.

[2] ACAR, G., JUAREZ, M., NIKIFORAKIS, N., DIAZ, C., GÜRSES, S., PIESSENS, F., AND PRENEEL, B. FPDetective. *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13* (2013), 1129–1140.

[3] ECKERSLEY, P. How unique is your web browser? In *International Symposium on Privacy Enhancing Technologies Symposium* (2010), Springer, pp. 1–18.

[4] ENGLEHARDT, S., AND NARAYANAN, A. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 1388–1401.

[5] ENGLEHARDT, S., AND NARAYANAN, A. Online Tracking: A 1-million-site Measurement and Analysis. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, 1 (2016), 1388–1401.

[6] FAIZKHADEMI, A., ZULKERNINE, M., AND WELDEMARIAM, K. Fpguard: Detection and prevention of browser fingerprinting. In *IFIP Annual Conference on Data and Applications Security and Privacy* (2015), Springer, pp. 293–308.

[7] FIFIELD, D., AND EGELMAN, S. Fingerprinting web users through font metrics. In *International Conference on Financial Cryptography and Data Security* (2015), Springer, pp. 107–124.

[8] GÓMEZ-BOIX, A., LAPERDRIX, P., AND BAUDRY, B. Hiding in the Crowd: an Analysis of the Effectiveness of Browser Fingerprinting at Large Scale. In *WWW 2018: The 2018 Web Conference* (Lyon, France, Apr. 2018).

[9] HANNAK, A., SOELLER, G., LAZER, D., MISLOVE, A., AND WILSON, C. Measuring Price Discrimination and Steering on E-commerce Web Sites. *Proceedings of the 2014 Conference on Internet Measurement Conference - IMC '14* (2014), 305–318.

[10] LAPERDRIX, P., BAUDRY, B., AND MISHRA, V. Fprandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In *International Symposium on Engineering Secure Software and Systems* (2017), Springer, pp. 97–114.

[11] LAPERDRIX, P., RUDAMETKIN, W., AND BAUDRY, B. Mitigating Browser Fingerprint Tracking: Multi-level Reconfiguration and Diversification. *Proceedings - 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015* (2015), 98–108.

[12] LAPERDRIX, P., RUDAMETKIN, W., AND BAUDRY, B. Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints. *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016* (2016), 878–894.

[13] LERNER, A., SIMPSON, A. K., KOHNO, T., AND ROESNER, F. Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016. *Usenix Security* (2016).

[14] MOWERY, K., AND SHACHAM, H. Pixel Perfect : Fingerprinting Canvas in HTML5. *Web 2.0 Security & Privacy 20 (W2SP)* (2012), 1–12.

[15] NIKIFORAKIS, N., JOOSEN, W., AND LIVSHITS, B. PriVaricator. *Proceedings of the 24th International Conference on World Wide Web - WWW '15* (2015), 820–830.

[16] NIKIFORAKIS, N., KAPRAVELOS, A., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. *Proceedings - IEEE Symposium on Security and Privacy* (2013), 541–555.

[17] SAITO, T., TAKAHASHI, K., YASUDA, K., ISHIKAWA, T., TAKASU, K., YAMADA, T., TAKEI, N., AND HOSOI, R. OS and Application Identification by Installed Fonts. *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)* (2016), 684–689.

[18] STAROV, O., AND NIKIFORAKIS, N. XHOUND: Quantifying the Fingerprintability of Browser Extensions. In *S&P* (2017).

[19] TAKEI, N., SAITO, T., TAKASU, K., AND YAMADA, T. Web Browser Fingerprinting Using only Cascading Style Sheets. *Proceedings - 2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications, BWCCA 2015* (2016), 57–63.

[20] TORRES, C. F., JONKER, H., AND MAUW, S. Fp-block: usable web privacy by controlling browser fingerprinting. In *European Symposium on Research in Computer Security* (2015), Springer, pp. 3–19.

[21] VASILYEV, V. fingerprintjs2: Modern & flexible browser fingerprinting library, Aug. 2017. original-date: 2015-02-11T08:49:54Z.

[22] VASTEL, A., LAPERDRIX, P., RUDAMETKIN, W., AND ROUVOY, R. Fp-stalker: Tracking browser fingerprint evolutions. In *IEEE S&P 2018-39th IEEE Symposium on Security and Privacy* (2018), IEEE, pp. 1–14.

[23] YU, Z., MACBETH, S., MODI, K., AND PUJOL, J. M. Tracking the trackers. In *Proceedings of the 25th International Conference on World Wide Web* (2016), International World Wide Web Conferences Steering Committee, pp. 121–132.

Notes

[1] Augur: `https://www.augur.io`
[2] Ghostery: `https://www.ghostery.com`
[3] NoScript: `https://noscript.net`
[4] AdBlock: `https://getadblock.com`
[5] Privacy Badger: `https://www.eff.org/fr/privacybadger`
[6] Canvas Blocker: `https://github.com/kkapsner/CanvasBlocker`
[7] Canvas Defender: `https://multiloginapp.com/canvasdefender-browser-extension`
[8] Brave: `https://brave.com`
[9] Ultimate User Agent: `http://iblogbox.com/chrome/useragent/alert.php`
[10] Random Agent Spoofer: `https://github.com/dillbyrne/random-agent-spoofer`
[11] FP-Scanner dataset: `https://github.com/Spirals-Team/FP-Scanner`
[12] UA Parser: `https://github.com/ua-parser/uap-python`
[13] Modernizr: `https://modernizr.com`
[14] Caniuse: `https://caniuse.com`
[15] List of available features per browser: `https://github.com/Fyrd/caniuse/blob/master/data.json`
[16] Mutation observer API: `https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver`
[17] TCP fingerprinting: `http://lcamtuf.coredump.cx/p0f3`