# FPRandom: Randomizing core browser objects to break advanced device fingerprinting techniques

Pierre Laperdrix, Benoit Baudry, Vikas Mishra

# FPRandom: Randomizing core browser objects to break advanced device fingerprinting techniques

Pierre Laperdrix[1], Benoit Baudry[2], and Vikas Mishra[3]

[1] INSA de Rennes & INRIA, Rennes, France
pierre.laperdrix@insa-rennes.fr
[2] INRIA, Rennes, France
benoit.baudry@inria.fr
[3] Birla Institute of Technology & Science, Goa, India
vikasmishra95@gmail.com

**Abstract.** The rich programming interfaces (APIs) provided by web browsers can be diverted to collect a browser fingerprint. A small number of queries on these interfaces are sufficient to build a fingerprint that is statistically unique and very stable over time. Consequently, the fingerprint can be used to track users. Our work aims at mitigating the risk of browser fingerprinting for users privacy by 'breaking' the stability of a fingerprint over time. We add randomness in the computation of selected browser functions, in order to have them deliver slightly different answers for each browsing session. Randomization is possible thanks to the following properties of browsers implementations: (i) some functions have a nondeterministic specification, but a deterministic implementation; (ii) multimedia functions can be slightly altered without deteriorating user's perception. We present FPRandom, a modified version of Firefox that adds randomness to mitigate the most recent fingerprinting algorithms, namely canvas fingerprinting, AudioContext fingerprinting and the unmasking of browsers through the order of JavaScript properties. We evaluate the effectiveness of FPRandom by testing it against known fingerprinting tests. We also conduct a user study and evaluate the performance overhead of randomization to determine the impact on the user experience.

## 1 Introduction

Browser fingerprinting has reached a state of maturity where it is now used by many companies alongside cookies to identify and track devices for a wide range of purposes from targeted advertising to fraud prevention. Several studies have shown the growth of this technique along the years with both the discovery of new attributes and its spread on the web [1,2,18]. Englehardt et al. notably showed in 2016 that "the number of sites on which font fingerprinting is used and the number of third parties using canvas fingerprinting have both increased by considerably in the past few years" [8]. Third-parties are starting to turn to

the most recent browser APIs to collect as much device-specific information as possible. Olejnik et al. demonstrated that recent APIs like the Battey API [20] or the Ambient Light Sensor API [21] can leak device-specific information. Englehardt et al. discovered that some online scripts use the AudioContext API to get data on the audio capabilities of a device [8]. However, in the fast evolving landscape of Web standards, only a very low number of works have explored approaches to mitigate or thwart tracking through fingerprinting. The main challenge in designing a good defense is to preserve the user experience while sending information that will render device identification impossible. Since most of the different attributes that constitute a fingerprint are essential for browsing the web, changing a single one incorrectly can result in the complete loss of functionalities and the user is then unable to get access to the desired service. In this paper, we explore the use of browsers' flexibility to prevent tracking through advanced fingerprinting techniques along with an implementation in a Firefox browser called FPRandom.

**Key insight** Several API functions used to build a browser fingerprint are unnecessarily deterministic and provide device-specific information as a side effect. Browsers present untapped flexibility that can be exploited to obtain a constantly changing fingerprint. We especially investigate the two following areas:

- Browsers' JavaScript engines make deterministic implementation choices, even for some functions which are meant to be non-deterministic according to the ECMAScript. We show that by removing some of these deterministic coding choices, we can prevent the leakage of device-specific information.
- Randomness can be used to diversify how HTML multimedia elements are rendered in the browser. Leveraging human's perception of colors and sound, we introduce controlled noise into the rendering process of canvas and audio elements. Consequently, fingerprinting scripts constantly collect new values, but we preserve the user experience.

By introducing randomness into key browser subroutines, we break the stability of a fingerprint without artificially replacing values with pre-existing ones. This hampers user tracking since a server cannot rely on the stability of the fingerprint.

**Implementation and Evaluation** The landscape of browser fingerprinting has greatly evolved in the past few years with the push of HTML5 by developers to make the web more secure and more complete. The lists of plugins and fonts used to be two key attributes to identify devices, but they are now slowly becoming remnants of the past. Yet, more recent techniques such as canvas fingerprinting [1], AudioContext fingerprinting [8] and the unmasking of the browser through JavaScript properties' order [18] offer strong foundations for effective fingerprinting. Our work focuses on these three techniques. For each technique, we look at the official JavaScript specification with a special attention for the inner-workings of the targeted HTML elements. Then we design behavioural modifications that can impact the fingerprinting process without altering the user experience. We developed a working prototype based on Firefox called FPRandom and we evaluate both its performance and its resilience

against known fingerprinting scripts. The performed benchmarks show that the overhead introduced by our solution is negligible and that all scripts using the targeted attributes are affected.

The paper is organized as follows. Section 2 gives an overview of our approach along with details on existing techniques to break the linkability of fingerprints. Section 3 details the FPRandom browser with the different fingerprinting techniques we target. Section 4 provides an evaluation of the browser's performance and its ability to deceive known fingerprinting scripts. Section 5 discusses the related work while Section 6 concludes this paper.

## 2 Breaking linkability of fingerprints

Fingerprint uniqueness and stability are the two key properties that make browser fingerprint tracking a reality. In this work, we aim at breaking the stability of fingerprints so that collected fingerprints are not *linkable*. Diversity is a strong solution as constantly-changing information can be sent to fingerprinters without impacting the user experience. Below, we present two state of the art solutions that specifically target the linkability of fingerprints before we detail our approach.

### 2.1 Current solutions

**Blink** In a previous study, we explored with *Blink* the use of dynamic software reconfiguration "to establish a moving target defense against browser fingerprint tracking" [13]. By leveraging virtualization and assembling components from a large pool of operating systems, browsers, plugins and fonts, *Blink* is able to build environments that exhibit very different fingerprints over time. The strong advantage of this solution is that the generated browsing platforms are genuine: they do not present any inconsistencies contrary to the spoofing extensions discussed in Section 5. However, its major drawback is the complete machinery involved in the synthesis of these browsing platforms. To exhibit diverse fingerprints, *Blink* relies on a pool of hundreds of components that can occupy a large amount of disk space and the use of virtual machines can be very costly on less powerful computers. Moreover, *Blink* requires the use of several different browsers to increase its success rate at breaking fingerprint linkability. It is a sacrifice that not all users may be willing to make to increase their online privacy.

**PriVaricator** Nikiforakis et al. explore with *PriVaricator* how randomization can be used to render browser fingerprints unreliable for tracking [18]. The main insight of their study is that making fingerprints non-deterministic on multiple visits makes them hard to link across browsing sessions. They state that "creatively misrepresenting or lying about [collected] values introduces an element of non-determinism" and that "subtly misrepresenting key properties of the browser environment goes a long way towards combating fingerprinters". As

a proof of concept, they introduced into Chromium the concept of *randomization policies*, protection strategies that are activated when specific requirements are met. They designed policies against the collection of offset measurements of HTML elements and the enumeration of the list of plugins.

### 2.2 Our approach: exploiting browsers' flexibility

In this work, we propose to exploit browsers' untapped flexibility to introduce randomness. Instead of changing software components at runtime like Blink or lying on specific values like PriVaricator, we want to increase non-determinism in browsers to reduce side-effects that cause fingerprintable behaviors.

**Flexibility of the implementation of the JavaScript specifications** The official ECMAScript specification, the de facto standard for scripting language on the web, allows some flexibility in actual JavaScript implementations. Different parts of the specification give some leeway by clearly indicating that specific choices are left for the implementation. The ECMA organization strictly codifies the interpretation of the language but the exact details of how it works remain in the hands of browser vendors. For example, as we will see in the next Section, the enumeration order of JavaScript properties are not detailed by the ECMAScript specification but each browser presents its own unique order. Developers have made deterministic choices when they implemented these functions. By taking a step back and removing what can be interpreted as a surspecification of the standard, we are able to thwart fingerprinting vectors that rely on these detectable side-effects.

**Flexibility of the renderings of multimedia elements** Vendors are constantly striving to improve their browsers to provide the latest innovations and the best possible experience to their users. Changing some attributes collected in a browser fingerprint like the user agent or the screen resolution can negatively impact how a page is displayed to the detriment of users. However, the rendering of HTML multimedia elements can be made more flexible and less deterministic without degrading the user experience. Especially, we can exploit users' perception of color and sound to introduce imperceptible noise that impacts the stability of specific browser routines. The key challenge here is to apply very small modifications that no user will notice while a fingerprinting script will output constantly changing values at every execution.

## 3 Implementation

To experiment with randomization, we target three of the most recent fingerprinting techniques: canvas fingerprinting as it is a prime example of a dynamic media element and "in the top 5 of the most discriminating attributes" [14]; the Web Audio API recently observed in fingerprinting scripts by Englehardt et al. [8]; the leakage of system information through JavaScript properties' order found by Nikiforakis et al. in the Cookieless Montster study [18]. All the mitigation techniques detailed in this section are implemented in a modified version

of Firefox called FPRandom. The complete patch for Firefox 54 is available on GitHub [4] along with a fully-compiled prototype for Linux systems.

### 3.1 Canvas API

**Definition** Canvas fingerprinting was firstly introduced by Mowery et al. [16] and observed on the Internet by Acar et al. [1]. Its goal is to use the Canvas API of a browser to draw an image that can help differentiate one device from another. Each device executes the exact same set of instructions and depending on both hardware and software, rendered images present variations. Figure 1 shows the canvas test we run on the AmIUnique website. The test consists in displaying two lines of text with different shapes merged together. Here, depending on the hardware and the installed drivers, the rendering of shapes and colors slightly vary between devices. Then, depending on the software and most especially on the list of installed fonts, the lines of text can present great differences. In our previous study [14], we showed that one of the strength of canvas fingerprinting is its stability and that it is "in the top 5 of the most discriminating attributes". It is notably the "second highest source of entropy for mobile devices".

**Modification** The first modification we made to the Firefox source code is to introduce randomness inside the *ParseColor* function of the *CanvasRenderingContext2D* class. Every time a color is set inside a canvas fingerprinting script, the browser changes the actual RGB values of the parsed color by adding or removing a very small number for each color channel. For example, if a script asks to draw an orange rectangle, the browser will paint the canvas element as requested. However, for every browsing session, the browser will use a slightly different orange than the last time. Modifying the *ParseColor* method enables us to support the full range of color declaration (for example, you can chose a color by directly setting its RGB values or you can simply write its name like "gold" or "orange"). The impact on the user experience is almost non-existent as the difference in color is very hard to see with the naked eye. Finally, it should be noted that we differentiate ourselves from tools called "canvas poisoners" that change the RGB values of each pixel of the rendered image independently from one another. Mowery et al. wrote that they are not "a feasible defense" against current scripts because the noise can be lifted by repeating a test a few times and comparing the results. They add that the aggressive introduction of noise "degrades the performance of <canvas> significantly for legitimate applications"[16]. With our approach, the color on a whole surface is consistent as we do not introduce random noise on separate pixels. As discussed in Section 3.4, we can apply the exact same modification for every run of a fingerprinting script. If a fingerprinter were to repeat the same canvas test more than once, he will not be able to notice differences whereas canvas poisoners present noticeable variations between runs. The second modification operates in the *SetFont* function of the *CanvasRenderingContext2D* class and changes a font set by a script by one present on

[4] `https://github.com/plaperdr/fprandom`

the operating system. For the scripts asking for a fallback font, the stability is broken as a font different from the previous session will be presented.

**Example** Figure 2 illustrates the impact of FPRandom on the exact same canvas test with all protection features enabled. The blue, orange and green colors are slightly different for each run and the used fonts are chosen among the ones present on the operating system. The more fonts are installed on the user's system, the bigger the diversity of generated canvas renderings will be. By changing at runtime core properties of elements present in a canvas test, we break the stability of this technique while still preserving the user experience.

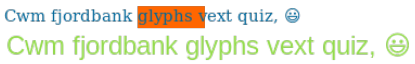



Fig. 1: Original canvas rendering with standard colors and the default fallback font

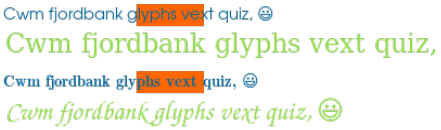



Fig. 2: Canvas renderings with modified colors and fonts

### 3.2 AudioContext API

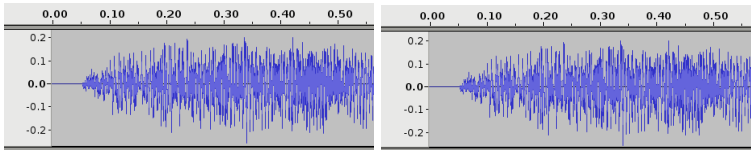
**Definition** Discovered by Englehardt et al. while crawling the web looking for trackers [8], AudioContext fingerprinting is a newcomer in the browser fingerprinting domain. The AudioContext API provides an interface to create a pipeline to process audio. By linking audio modules together, you can generate audio signals and apply very specific operations like compression or filtering to generate a very specific output.

In audio, sampling is applied to convert a continuous signal into a discrete one. This way, a computer can easily process audio in distinct blocks called frames. Each frame is composed of samples that represent the value of the audio stream at a specific point in time. Englehardt et al. have shown that, depending on the audio stack of your system (both software and hardware), the exact value of each of these frames slightly vary between devices. An audio fingerprint can then be created similarly to what is done with the Canvas API.

**Modification** We performed an analysis of audio fingerprints that we collected on AmIUnique.org and the results can be found in Appendix A. We decided to introduce very small noises directly into the audio processing routines of the browser so that tests using any number of AudioContext modules are all impacted. We operate at the heart of the *AudioBuffers* of the *AudioNodeEngine* as they contain the frames of the processed audio. By modifying key functions, we slightly decrease the volume of processed buffers by a factor ranging between 0.000 and 0.001. This way, a frame can present very small variations where only the smallest decimal digits are affected. With the use of very small factors, it is impossible to detect modified sections from unmodified ones just by listening to the rendered track as the differences between the original and modified track can genuinely be interpreted as side effects or simple noise of the whole audio stack

of the device. For fingerprinting scripts, these modifications produce a different hash as the audio routine will be ever so slightly different for each browsing session.

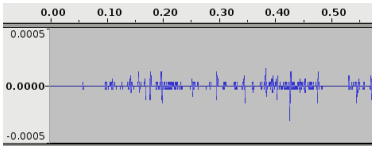
**Example** Figure 3 shows three waveforms of the first second of the "Ride of the Valkyries" from Wagner. The audio pipeline we set up for this example performs two operations. It first increases the volume of the track with a *GainNode* and then compresses it through a *DynamicsCompressorNode*. The waveform in Figure 3a represents the output from an unaltered pipeline and the one in Figure 3b from a pipeline with our volume modification. The last waveform in Figure 3c represents the difference between the first two (i.e. the introduced noise). In order to see the impact of FPRandom, the 3rd waveform has been zoomed in at 1000%. The scale is a clear indication that the generated noise is inaudible, proving that the impact on the user experience is non-existent audio wise but it still impacts the created audio fingerprint.




(a) Original waveform




(b) Modified waveform




(c) Difference between the two waveforms

Fig. 3: Visualization of audio rendered through the AudioContext API

### 3.3 Order of JavaScript object's properties

**Definition** By analyzing the content of JavaScript objects in the browser, Nikiforakis et al. discovered that "the order of property-enumeration of special browser objects, like the navigator and screen objects, is consistently different between browser families, versions of each browser, and, in some cases, among deployments of the same version on different operating systems"[18]. This way, if someone were to hide the true browser's identity, enumerating the properties of a special object would simply unmask it. As stated by the latest ECMAScript Language Specification ratified in June 2016, "mechanics and order of enumerating the properties is not specified" (see section 13.7.5.15 EnumerateObjectProperties of [7]). This ordering behavior is entirely dependent on the browser's implementation. Chrome and Firefox yield vastly different enumeration orders for native objects like *navigator*. For non-native JavaScript objects, both browsers first return integers in ascending order and then strings in insertion order. This choice

is arbitrary and many developers have long debated for the best and most logical behavior as illustrated by this long discussion on the V8 bug tracker[22].
**Modification** The browser's unmasking added by the surspecification of the ECMAScript standard can simply be undone by modifying the *jsiter* class of Firefox. A special flag called "JS_MORE_DETERMINISTIC" can be activated at compile time to sort IDs for both native and non-native objects in a deterministic way. By tweaking the behavior of the *SortComparatorIds* structure used with this special flag, we flip its purpose by not making the JavaScript engine more deterministic but by generating a unique enumeration order every time the browser is launched.

With the "JS_MORE_DETERMINISTIC" flag activated, the enumeration of a JavaScript object first returns integers in ascending order and then strings in alphabetical order. By diving even deeper into the source code, we found that the string comparison done by the browser relies on the "Latin-1" or "ISO/CEI 8859-1" encoding of each string. When comparing two strings, the engine goes through one character at a time and performs a simple subtraction of their code points (i.e. their place in the Latin-1 character set, see [27]) to determine which character is in front of the other. When a difference is detected, the engine knows how to order the two strings as the result is either positive or negative. Appendix B gives an example of such comparison between the *appName* and *appVersion* strings.

In order to change the enumeration order for each browsing session, we assign a random order for each combination (i.e. for each possible subtraction result) from the Latin-1 character set. As the first code point starts at position n°32 and the last one is at n°255, we generate in total 223 different booleans to cover all possible combinations. Any attempt to unmask the browser through this technique is then prevented.

### 3.4 Randomization strategy

All the modifications described in this section can be executed in different ways when browsing the web. Here, we detail the two randomization strategies present in FPRandom while discussing their own strengths and weaknesses.
**Random mode** The first mode that we propose in FPRandom is the "Random" strategy. Every time the modified functions are executed in the browser, they will return random values. The advantage is that it prevents cross-domain tracking as two scripts on two different domains (even from the same provider) would collect different values on both sites. However, the major downside of this solution is that it presents "transparency" problems as discussed by Nikiforakis et al. in the PriVaricator study. [17]. If a fingerprinter were to study the presence of randomness, a script could execute the same test several times to detect instability. Depending on the test, a statistical analysis could be performed to reduce or remove the introduced randomness but it requires far more means and a certain insight into the way noise is introduced to get meaningful results. The "Random" mode is the default one in FPRandom as we have no recorded proof of such behaviors from today's major fingerprinting actors.

**Per session** The second mode initializes all the randomized variables at startup and they are never modified on subsequent executions. The advantages of this strategy is that it cannot be detected through repeated measurements as the browser will always return the same answers for an identical fingerprinting test. The downside is that it only breaks linkability between browsing sessions as the same fingerprint will be presented to all websites until the browser is rebooted.

## 4 Evaluation

### 4.1 Deceiving fingerprinting scripts

As pointed out by [17], while it is possible to analyze the JavaScript code that runs inside the browser and detect fingerprinting scripts, it is much more complicated to find fingerprinters that can act as black-box oracles for our work. Some websites give a specific identifier associated with a device's fingerprint but others map collected attributes in a very specific way that is confidential and that is entirely performed on the side of the server. The main challenge in assessing the impact of FPRandom is to find fingerprinting scripts that use the advanced techniques we target and retrieve the right information (either an identifier or the fingerprint data that is sent).

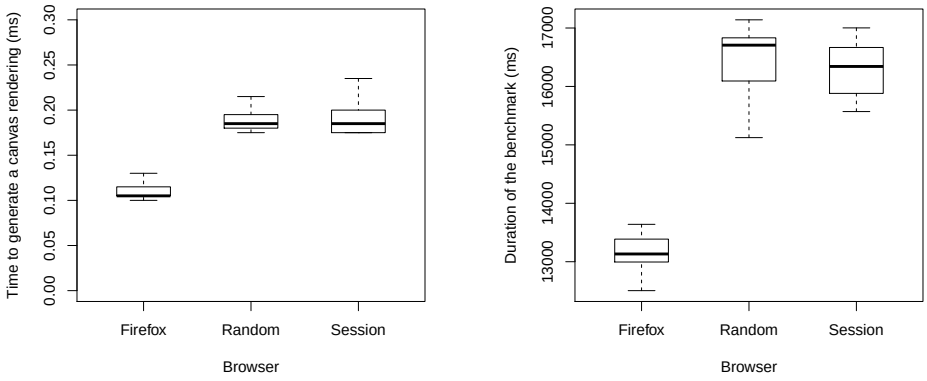
**Fingerprintjs2** is the second version of a popular open-source fingerprinting library that collects 25 different attributes and hash them all into a single value [9]. We executed the complete test suite of this library 100 times on both a standard version of Firefox 54 and FPRandom. On Firefox 54, we obtained the same hash for all of the 100 executions. For FPRandom, we collected 100 different ones with the Random mode and a single one in Session mode. These results show how fragile the test suite is for identification. The introduction of noise on a single attribute is sufficient to be considered as a "new" fingerprint.

**Maxmind** is a company specialized in IP geolocation and online fraud prevention. As part of its fraud detection system, Maxmind has a "device tracking add-on" to identify devices "as they move across networks" [15]. The main add-on script sends the complete device fingerprint at a specific address in a POST request. We manually analyzed the code of the add-on and found that it collects the rendering of a canvas test along with the enumeration order of both the *navigator* and *screen* objects. After 100 runs of the fingerprinting script, FPRandom gives a different canvas hash at each execution whereas a standard Firefox build always send the same result. For the enumeration orders, the behavior of Firefox 54 is the expected one and returns the exact same order for both JavaScript objects. For FPRandom, the browser gives a unique and different enumeration order at each session.

**Limitations** Our approach does not deal with static attributes like the user-agent or the timezone but it can mitigate the collection of dynamic attributes from APIs like Canvas or Battery. Scripts that do not rely on the attributes we target can still build their own browser fingerprint and use it for tracking, albeit with a less complete view of the user's system.

### 4.2 Performance

We use three different JavaScript benchmark suites to assess the performance overhead introduced by FPRandom. The experiments were conducted on a laptop running Fedora 25 with an Intel Core i7-4600U CPU @ 2.10GHz. The tests were performed using Firefox 54 (Nightly version) with and without our modifications present and enabled.




(a) Canvas API (b) Audio API

Fig. 4: Benchmarking results

**Canvas** As there are no benchmarks that specifically target the Canvas API, we developed our own test to asses the overhead introduced by our color and font variations. We repeated the test shown in Figure 1 1,000 times and measured the time it takes for each image to be fully rendered inside the browser. To get precise measurements, we used the JavaScript Performance API that provides timestamps with an accuracy up to the microsecond [23]. Figure 4a illustrates the difference between a vanilla version of Firefox 54 and FPRandom. While an unmodified version takes 0.12ms to render the image, our modified browser is about 0.06ms longer on average in both randomization modes. This difference corresponds to the time it takes to properly choose a new font and introduce variations in the canvas colors. With these reported numbers, we consider the overhead here to be negligible as a rendering time of less than a single millisecond will not impact the user experience.

**AudioContext** To assess the impact of FPRandom on the AudioContext API, we use a WebAudio benchmark developed by a Mozilla developer [4]. The benchmark performs a series of 19 different tests from simple gain operations to more complex mixing procedures. Each test instantiates an *OfflineAudioContext* object which is one of the objects we targeted when modifying the Firefox source code. The results in Figure 4b indicate the time it takes to perform the complete series of 19 tests. It should be noted that we repeated the test suite 30 times. The mean value for FPRandom is about 25% higher than its Firefox counterpart. The "Random" mode is also a little longer than the "Session" one. This increase can be explained by the fact that the modified code is executed a very

high number of times in a single test. By comparison, the modification made to the canvas API is only executed once for a test. We instrumented the browser to find a precise number and we found out that a single run of the benchmark enters our modified function more than 8,862,000 times. As a point of reference, the AudioContext test found by Englehardt et al.[8] only enters our function less than 3,000 times. With these numbers, we qualify the benchmark as extremely intensive. The increase in processing time may not be visible for less extreme and more traditional uses of the API. We leave for a future work the exploration of different implementation strategies where only a specific percentage of audio frames would be modified, leading to an increase in performance.

**JavaScript enumeration order** As a final performance test for FPRandom, we decided to run a standard JavaScript benchmark to see if the modification made on the enumeration order has an impact on the overall JavaScript engine. We used the recent JetStream benchmark[12] which is developed as part of the WebKit browser engine. Currently, the 1.1 version performs 39 different tests and covers a wide variety of advanced workloads and programming techniques. It integrates tests from well-known benchmarking suites like SunSpider or Octane along with new ones developed specifically for JetStream (more details [11]). The

Table 1: JetStream benchmark results

| | Firefox | FPRandom-Random | FPRandom-Session |
|---|---|---|---|
| Latency | 76.075 ± 1.3250 | 74.553 ± 1.8074 | 74.767 ± 1.2530 |
| Throughput | 251.97 ± 3.2912 | 252.32 ± 2.4214 | 256.02 ± 1.1213 |
| Total | 147.45 ± 1.5753 | 146.23 ± 1.9204 | 147.61 ± 1.1257 |

results are present in Table 1 (the bigger the score, the better the performance). As we can see, the scores are almost identical and no real distinction can be made between Firefox and FPRandom. The behavior of the two browsers are similar on both JavaScript throughput and latency and the impact of our order modification is seemingly nonexistent.

**Web crawl** In order to assess more globally the impact of our modifications on day-to-day browsing, we crawled the thousand most popular websites as reported by Alexa [3] on both a vanilla version of Firefox 54 and FPRandom. We used Selenium as the engine for the crawl of both browsers, and we used the Performance API of the browser to measure the time it takes for the DOM of the main page to be completely loaded. Specifically, we used the *domLoading* and *domContentLoadedEventStart* events to make our measurements as they are independent of problems related to network instabilities and congestion. Because of server unavailability and certificate problems encountered during our crawl, we miss loading times for 43 sites. The results can be found in Table 2. In general, load times are extremely close between a vanilla Firefox and FPRandom. Mean times indicate a slightly better performance for Firefox. Yet, in both cases, the standard deviation is very high, meaning that the collected loading times are very dispersed between 0 and 5s. These numbers demonstrate that the modifications introduced in FPRandom do not have a visible impact on the user experience for day-to-day browsing. Moreover, we can also say that the amount

of site breakage is kept to a minimum as only a single script provided us with an error due to our enumeration modification. The modifications on both the Canvas and AudioContext API had no impact on site breakage.

Table 2: Web crawl results

| | Times collected | Min (ms) | Max (ms) | Mean (ms) | SD (ms) |
|---|---|---|---|---|---|
| Firefox | 957 | 10 | 64728 | 1602 | 3712 |
| FPRandom | 958 | 9 | 55852 | 1823 | 3935 |

### 4.3 User study

An important aspect of FPRandom is that it modifies multimedia elements that can be seen or heard by the user. To make sure that the modified subroutines do not degrade substantially the user experience at the cost of better privacy, we ran a preliminary user study in February 2017. Its goal was to compare multimedia elements as rendered by a normal version of Firefox 54 with modified ones rendered by FPRandom. The study was divided into two phases: the first was focused on the modifications made to canvas elements while the second investigated the impact on the AudioContext API. The link to our survey was communicated through regular channels like social networks and team mailing lists. We received an answer from 20 participants and the results are as follows:

- Half of them noticed a color difference between the original canvas rendering and the 10 modified ones, the other half did not.
- 70% said that some fonts made the strings harder to read and only one person said that it was significantly harder for all of the modified renderings.
- For the AudioContext API, only 25% detected a difference between the original track and the 3 modified ones.
- For people who heard a difference, they all qualified the difference with words like “tiny” or “small”.

These results give us confidence in the direction we took with our approach but we plan on conducting a more thorough study to pinpoint more precisely avenues for improvement with more participants. Still, we will investigate how we can exclude some exotic fonts as they can make strings in canvas renderings harder to read for users.

## 5 Related work

While we focus in this paper on breaking the linkability of browser fingerprints, other approaches have been designed to tackle fingerprint tracking.

**Blocking extensions** Several works report that one strong way to prevent tracking is to block scripts before they are loaded by the browser [1,8,16]. Browser extensions like NoScript[19], Ghostery[10] or Disconnect[6] are illustrations of such solutions. However, these extensions require complete lists of

all scripts performing fingerprinting to correctly block them. As the web is constantly evolving and actors in the tracking business are changing, it is very hard to maintain up-to-date lists to protect users' privacy.

**Spoofing extensions** Another way to mitigate tracking based on browser fingerprints is to return incorrect information to trackers. Dozens of spoofing extensions already exist for both Chrome and Firefox. With very few steps, a Firefox browser can easily report that it is a Chrome one and vice versa. However, as shown by Nikiforakis et al. [18], these solutions produce inconsistent fingerprints. While the user agent reports one information, a JavaScript property will tell a different story, proving that the browser has deliberately changed its default values. Torres et al. follow in the footsteps of spoofers and created a solution called FP-Block that is an implementation of the concept of separation of web identities [25]. The idea is simple: FP-Block generates a fingerprint (i.e. an identity) for each website that the browser is in contact with. Every time the browser reconnects to the same website, it will reuse the generated identity. As a defense against fingerprinting, the premise is great but unfortunately, the implementation presents the same shortcomings as the extensions stated above. Generated fingerprints are inconsistent and it is easy to find supposedly hidden information as there is an incomplete coverage of methods used for fingerprinting.

**Use of multiple browsers** One approach to obtain different browser fingerprints is to use multiple browsers. However, Boda et al. showed the existence of *Cross-browser fingerprinting* [5]. By collecting enough OS-specific data like the list of fonts or plugins, a script can identify a device behind multiple browsers as these information are stable from one browser to the next. A more recent method by Cao et al. can identify with high precision users across different browsers through the use of the WebGL API [28].

**Tor browser** The Tor browser is a modified Firefox specifically designed for the Tor network. Its approach towards fingerprinting is to have fingerprints that are as uniform as possible. As stated by the Tor design document [24], its defenses include "value spoofing", "Subsystem Modification or Reimplementation", "Virtualization", "Site Permissions" and "Feature or Functionality Removal". However, this approach poses several problems. First, by design, fingerprints from Tor browser are very specific and thus already known to trackers. Its users can easily be identified as using the Tor browser. Then, the offered protection is extremely brittle since a simple change can change the standard fingerprint to a unique one. To remain effective, customizability and personalization are severely hampered because of this mono-configuration. Finally, by blocking specific browser APIs, the Tor browser restricts users from benefiting from the full array of browser features that could enrich their browsing experience.

## 6 Conclusion and future perspectives

In this work, we aim at breaking the stability of browser fingerprints over time to improve users' privacy. By identifying APIs with restrictive implementation with respect to the JavaScript specification, we introduce randomness to produce

slight variations at each execution of a fingerprinting script. We also generate noise inside HTML multimedia elements to alter their rendering without deteriorating user's perception. The approach presented in this work can be generalized to more fingerprinting vectors. For example, there exists other parts in the ECMAScript specification that leave the exact details of the implementation in the hands of developers. The use of Math constants in the browser can be used to unveil information about the device and its browser [26]. If we take a look at Section 20.2.2. of the official JavaScript specification [7], it is written that "the choice of algorithms is left to the implementation" and that the behavior of mathematical functions "is not precisely specified". This means that the actual libraries used for these functions could be diversified to prevent the unmasking of the operating system. In the end, the main challenge that remains here is to perform an exhaustive search to identify and anticipate future fingerprinting mechanisms. By locating key functions that could reveal device-specific information, we could preemptively introduce randomness to reinforce users' privacy on the web.

We also developed a working prototype called FPRandom that targets the following attributes of the browser fingerprinting domain: canvas fingerprinting, AudioContext fingerprinting and the unmasking of the browser through the order of special JavaScript objects. By looking at the specification and analyzing the browser's source code, we modified key locations to introduce very small noise that prevents the use of these fingerprinting vectors for identification. Our tests show that our modifications impact known fingerprinting scripts that use the targeted attributes. A careful attention was also given to preserve the user experience as much as possible and our performance benchmarks indicate that the introduced overhead is very small.

## References


1. G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS 2014)*. ACM, 2014.
2. G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. Fpdetective: dusting the web for fingerprinters. In *Proc. of the Conf. on Computer & Communications Security (CCS)*, pages 1129–1140. ACM, 2013.
3. Alexa - The top 500 sites on the web. `http://www.alexa.com/topsites`.
4. Benchmarks for the WebAudio API. `https://github.com/padenot/webaudio-benchmark`.
5. K. Boda, A. M. Földes, G. G. Gulyás, and S. Imre. User tracking on the web via cross-browser fingerprinting. In P. Laud, editor, *Information Security Technology for Applications*, volume 7161 of *Lecture Notes in Computer Science*, pages 31–46. Springer Berlin Heidelberg, 2012.
6. Disconnect's official webpage. `https://disconnect.me/`.
7. ECMAScript® 2016 Language Specification. `http://www.ecma-international.org/ecma-262/7.0/index.html`.

8. 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*, CCS '16, pages 1388–1401, New York, NY, USA, 2016. ACM.
9. fingerprintjs2, modern and flexible browser fingerprinting library, a successor to the original fingerprintjs. https://github.com/Valve/fingerprintjs2.
10. Ghostery's browser extension. https://ghostery.com/our-solutions/ghostery-browser-extension/.
11. Introducing the JetStream Benchmark Suite. https://webkit.org/blog/3418/introducing-the-jetstream-benchmark-suite/.
12. JetStream benchmark. http://browserbench.org/JetStream/.
13. P. Laperdrix, W. Rudametkin, and B. Baudry. Mitigating browser fingerprint tracking: multi-level reconfiguration and diversification. In *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015)*, Firenze, Italy, May 2015.
14. P. Laperdrix, W. Rudametkin, and B. Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *37th IEEE Symposium on Security and Privacy (S&P 2016)*, 2016.
15. Maxmind's Device Tracking Add-on for minFraud Services. http://dev.maxmind.com/minfraud/device/.
16. K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In M. Fredrikson, editor, *Proceedings of W2SP 2012*. IEEE Computer Society, May 2012.
17. 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*, pages 820–830. International World Wide Web Conferences Steering Committee, 2015.
18. N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proc. of the Symp. on Security and Privacy*, pages 541–555, 2013.
19. NoScript's official webpage. https://noscript.net/.
20. L. Olejnik, G. Acar, C. Castelluccia, and C. Daz. The leaking battery: A privacy analysis of the html5 battery status api. *IACR Cryptology ePrint Archive*, 2015:616, 2015.
21. L. Olejnik and A. Janc. Stealing sensitive browser data with the W3C Ambient Light Sensor API. https://blog.lukaszolejnik.com/stealing-sensitive-browser-data-with-the-w3c-ambient-light-sensor-api/.
22. Wrong order in Object properties interation - V8 bug tracker. https://bugs.chromium.org/p/v8/issues/detail?id=164.
23. High Resolution Time Level 2 (JavaScript Performance API. https://www.w3.org/TR/hr-time/#dom-domhighrestimestamp.
24. The Design and Implementation of the Tor Browser - Cross-Origin Fingerprinting Unlinkability. https://www.torproject.org/projects/torbrowser/design/#fingerprinting-linkability.
25. C. F. Torres, H. Jonker, and S. Mauw. *FP-Block: Usable Web Privacy by Controlling Browser Fingerprinting*, pages 3–19. Springer International Publishing, Cham, 2015.
26. Tor bug tracker - Math routines are OS fingerprintable. https://trac.torproject.org/projects/tor/ticket/13018.
27. Codepage layout - ISO/IEC 8859-1. https://en.wikipedia.org/wiki/ISO/IEC_8859-1#Codepage_layout.

28. S. L. Yinzhi Cao and E. Wijmans. (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In *Proceedings of the 2017 Network & Distributed System Security Symposium*, NDSS '17, 2017.

## A Analyzing differences in the AudioContext API

In order to have a better understanding of the diversity of audio fingerprints on the web, we deployed the AudioContext script found by Englehardt et al. on the AmIUnique.org website (used in our Beauty and the Beast study[14]). After discarding more than 1,000 fingerprints from browsers that did not implement the AudioContext API, we collected in total 19,468 audio fingerprints on a period of 100 days between June and September 2016. The results of this study can be found in Table 3. We use the Shannon entropy in bits to better represent the probability distribution of each of the attributes. The higher the entropy is, the more diversity is exhibited between devices.

Table 3: Study of 19,468 audio fingerprints

| Name | Entropy (bits) | Size of the biggest set | Number of distinct values | Number of unique values |
|---|---|---|---|---|
| acSampleRate | 1.18 | 9549 | 10 | 3 |
| acState | 0.99 | 10821 | 2 | 0 |
| acMaxChannelCount | 0.38 | 18580 | 11 | 1 |
| acNumberOfInputs | 0.0 | 19468 | 1 | 0 |
| acNumberOfOutputs | 0.0 | 19468 | 1 | 0 |
| acChannelCount | 0.0 | 19468 | 1 | 0 |
| acChannelCountMode | 0.0 | 19468 | 1 | 0 |
| acChannelInterpretation | 0.0 | 19468 | 1 | 0 |
| anFftSize | 0.0 | 19468 | 1 | 0 |
| anFrequencyBinCount | 0.0 | 19468 | 1 | 0 |
| anMinDecibels | 0.0 | 19468 | 1 | 0 |
| anMaxDecibels | 0.0 | 19468 | 1 | 0 |
| anSmoothingTimeConstant | 0.0 | 19468 | 1 | 0 |
| anNumberOfInputs | 0.0 | 19468 | 1 | 0 |
| anNumberOfOutputs | 0.0 | 19468 | 1 | 0 |
| anChannelCount | 0.99 | 10821 | 2 | 0 |
| anChannelCountMode | 0.0 | 19468 | 1 | 0 |
| anChannelInterpretation | 0.0 | 19468 | 1 | 0 |
| audioDynSum | 3.28 | 5698 | 53 | 5 |
| audioDynHash | 3.43 | 5697 | 72 | 12 |

Most of the collected attributes have a single value and do not provide any ground to distinguish one device from another. From the collected audio fingerprints, only 3 attributes have an entropy superior to a single bit:

– *acSampleRate* is the default sample rate of a created track when using the *AudioContext* API. The most common values are 44,1kHz (49,0% of col-

lected fingerprints) and 48kHz (48,5%) but some browsers still present some unusual ones (1,7% have 192kHz and 0,7% 96kHz).
- *audioDynSum* is the sum of 500 frames generated by a very specific audio processing (compressed audio from an oscillator). The precision of each frame is up to 15 decimal digits. The large majority of values are really close to each other with differences only appearing from the 6th or 7th decimal digit.
- *audioDynHash* is similar to *audioDynSum* as it takes the exact same output but it covers the entirety of the rendered track instead of a few hundreds frames. As it covers a larger space, the entropy is a little higher and this test exhibits more diversity than all other collected attributes.

With these results, we decided to focus only on the differences created by the audio processing performed inside audio nodes. Especially, we want to introduce random noise in the computed frames so that each run of the same test produces different variations. Other values like the default sample rate are still interesting to change but they can easily be modified and they are not the focus of this work.

## B Example of string comparison when ordering JavaScript properties

Figure 5 illustrates the comparison mechanism between the *appVersion* and the *appName* strings. The engine starts with the 'a' letter on both strings. Translating this letter to their corresponding Latin-1 code points yields the decimal numbers '97'. Subtracting 97 from 97 results in 0. As no difference is detected, the engine continues but faces the exact same result for both the second and third characters in each string as they are identical 'p' letters. However, the behavior is different from the fourth character. The first string presents a 'V' and the second an 'N'. Translating to their decimal code points yields '86' and '78'. This time, since the subtraction $86 - 78 = 8$ does not give a zero, it informs the engine that a difference has been detected. As the result is positive, *appName* is placed before *appVersion*. If the result of the subtraction were to be negative, it would have been the opposite order.

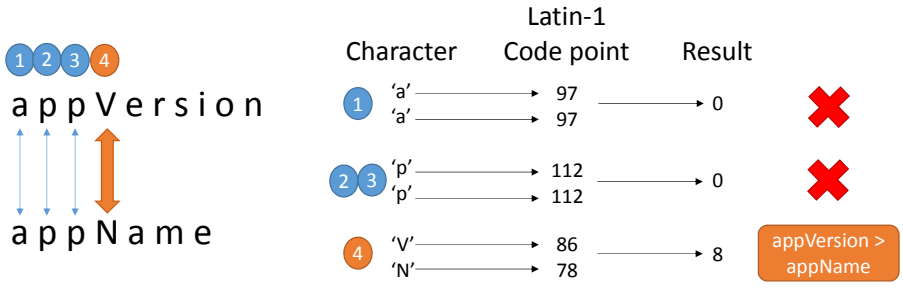



Fig. 5: String comparison between the *appName* and *appVersion* properties