6-2019

Preventing Browser Fingerprinting by Randomizing Canvas

Rianna Quiogue
Union College - Schenectady, NY

Preventing Browser Fingerprinting by Randomizing Canvas

By

Rianna Quiogue

* * * * * * * * *

Submitted in partial fulfillment

of the requirements for

Honors in the Department of Computer Science

UNION COLLEGE

June, 2019

Abstract

RIANNA QUIOGUE Preventing Browser Fingerprinting by Randomizing Canvas. Department of Computer Science, June, 2019.

ADVISOR: Matthew Anderson

Whether users know it or not, their online behaviors are being tracked and stored by many of the websites they visit regularly through a technique called browser fingerprinting. Just like a person's physical fingerprint can identify them, users' browser fingerprints can identify them on the Internet. This thesis outlines the techniques used in browser fingerprinting and explains how although it can be used for good, it can also be a major threat to people's online privacy and security. Since browser fingerprinting has gained popularity among many websites and advertising companies, researchers have been developing ways to counteract its effectiveness by creating programs that lie to fingerprinters or override a browser's innate properties in order to protect users' true identities. Our project proposes that by adding randomization to the `canvas` attribute in a Chromium browser, fingerprinting scripts will be rendered less effective. We compare our countermeasure (the canvas modifications) to a previous study, Privaricator [7], that focused on randomization in other attributes in Chromium. We reimplement Privaricator's modifications into the newest version of Chromium source code and implement our canvas modifications into a separate Chromium source code. We then test Privaricator and our countermeasure against several fingerprinters to obtain repeatability rates to determine and compare the success of each countermeasure. We also test both countermeasures against Panopticlick's online fingerprinting test to determine detectability of both countermeasures. We found that both countermeasures have the same repeatability rates when tested against fingerprinters, but Panopticlick was able to detect randomization in our countermeasure and not in Privaricator. We discuss future improvements to our countermeasure to potentially prevent detectability. We also discuss the effects on appearance of webpages, since canvas is a visible component on some websites.

Contents

List of Figures

List of Tables

1 Introduction

Many companies gather data on who visits their website including how many times and how often. One way of finding this information utilizes third-party cookies, which are pieces of information stored in the user's computer created by the website to track a user's browsing history. Cookies are often blocked by users and are easily detectable [3]. This motivated the need for a new way to track users online. Thus, browser fingerprinting was developed and has since been widely adopted. Browser fingerprinting gathers information accessible to the host website via Javascript about a user's browsers and operating systems, creates a unique "fingerprint" for each user. Browser fingerprinting runs in the background of a user's computer as part of the content of this webpage without their consent, therefore going unnoticed.

There are beneficial reasons for fingerprinting users, for example detecting fraud [8]. Websites that contain valuable, private information such as banking accounts or important documents use fingerprinting as a supplement to passwords and two-factor authentication to ensure that a user's identity is not being falsified or stolen. But on the other hand, fingerprinting can also be abused to log a user's history information across multiple websites or make it easier for cyber criminals to hack into a system once they know specific details about a user's digital device. A user's behavior, interests, and browsing history can be tracked and logged for a website or third-party's use, making ads or attacks more targeted to a specific user. Although many are not aware of what browser fingerprint is, studies show users find online behavioral advertising creepy and an invasion of privacy [10], [9]. Often times, even selecting private browsing mode and deleting all of one's history does not protect users from being uniquely identified by their fingerprint. Fingerprinting is quite deceiving because its tracking techniques are unseen by users. Unlike login credentials, search or find boxes, or user clicks, there is no visual evidence or interaction from the user in order for his or her information to be gathered. Fingerprinting happens in the backend using seemingly insignificant or generic attributes of the user's browser family, browser version, operating system, HTTP preferences, JavaScript settings, and Flash capabilities [2].

Eckersley [3] was one of the first to study device's vulnerability to browser fingerprinting. They created their own fingerprinting algorithm based on the configuration information that is accessible to all websites and gathered a sample of about 470,000 fingerprints from users who visited their website, Panopticlick.eff.org. This research project of the Electronic Frontier Foundation is widely cited by over 800 research papers, as a reference to how easy it is to identify unique browsers. Of the 470,000 browsers fingerprinted, they found that 94.2% of browsers with Flash and Javascript were unique. In this study they found that browser's fingerprints change over time; most of the time it occurs naturally, for example when upgrading browser version or operating system versions to the most current version. They were able to

show that 99.1% of the time they were able to link a browser's old fingerprint to their new one when the change was "natural" or not due to a countermeasure. Since the article was published in 2010, they have continued their research on improving their fingerprinting algorithm to be more robust against modern countermeasures and to use fingerprinting techniques to combat them. They emphasize that browser fingerprinting is a "worst case" scenario for privacy because unlike cookies, they are an identifier that cannot be deleted by the user unless a large enough change in configuration is able to break the fingerprinter. The overall strength of the paper comes from their ability to identify users with high confidence and track their changing fingerprints over time. They are now running version 3.0 of Panopticlicik on the same webpage and can be visited by users to check how identifiable they are.

It is important for web users to be aware that this practice is commonly used, how it is employed, and what they can do to curb it. Based off our research, it seems as though many prevention programs or techniques currently in play are flawed, not very effective, or even helpful for fingerprinters. Fingerprinters that detect a user has employed prevention techniques can create an even more unique fingerprint since prevention of fingerprinting is not a widely used practice yet. Acar et al. [1] was able to identify 16 new fingerprinting scripts and Flash objects, some of which were active in the top 500 Alexa websites, a list of the highest-performing websites based on reach and page views among Alexa users. These fingerprinting scripts include gathering information on attributes such as navigator properties `userAgent, language` window screen properties `height, width, colorDepth, pixelDepth` and font loading calls. Acar et al. [1] found new fingerprinting techniques that intend to erase evidence that a user was fingerprinted by removing fingerprinting scripts after they have been run. They showed that fingerprinting is much more prevalent than previously estimated. BlueCava (a third-party fingerprinting service) was found on 250 of the top Alexa websites and 404 sites on the Alexa top million websites fingerprint users on their homepages. It is concerning that users' information is being gathered without their consent from websites they visit frequently. Browser fingerprinting is a threat to the specific privacy issue of linkability, the ability of an attacker to distinguish whether two items of interested are linked (in this case our current identity to our identity in the past). Being identifiable online is not as much of an issue as being tracked online; even if fingerprinters can identify specific users, this practice is useless if they cannot watch people's behavior over time. If fingerprint trackers are able to link our current fingerprint to our previous fingerprint, they are able to trace our browsing history and behavior over an extended period of time. This is the main concern and motivation for uniquely identifying users.

One previous study, Privaricator [7], has been able unlink users' fingerprints by randomizing certain attributes in a Chromium browser. We implement Privaricator's code into the current version of Chromium and we use it to compare its effectiveness against our proposed countermeasure's. Privaricator was one of

the first studies to explore the method of spoofing fingerprinters by telling "little white lies." Previously developed countermeasures focused on lying about more common attributes such as `operating system` or `browser family`. They aim to spoof these attributes in order to make their fingerprint look like other people's fingerprints, or in other words to look less unique. These countermeasures often fail because fingerprinters check other attributes or browser functionalities to determine if that person is lying. It is obvious when someone is lying about these attributes because other components of one's fingerprint reveal the ground truth. The attributes Privaricator chose to randomize are not checked as often by fingerprinters, so the randomizations successfully deceive the fingerprinter into thinking the user is a different person every time.

1.1 Research Question

Over the past two terms we explored and answered the following research questions: Does randomizing the `canvas` attribute in the Chromium browser achieve a repeatability rate that is lower than previously found in the Privaricator study?

1. Does it bypass detection by fingerprinting trackers as measured by the Panopticlick fingerprinting test?

2. Does it cause negligible difference to the user's experience by significantly changing visual representations of webpages?

There is one significant characteristic that sets Privaricator apart from other countermeasures. Popular fingerprint spoofers aim to make a large number of users look the *same* by completely changing the truth of attributes; for example changing everyone's browser family to all appear as though they're using Firefox even if they are using a different browser. The reason why browser family or operating systems is considered a "big lie" is because these attributes affect many other aspects of the browser. Browser family determines what fonts and plugins can be installed. The value of one's browser family is also stored in multiple places, for example in the the version of Flash installed, in the `navigator` object and in the `userAgent` object. This is because the browser family determines many of the functionalities and directly affect usage. Other attributes, like `plugins` and `canvas` may reveal information about a browser, but do not affect other attributes within the browser. In other words, the relationship between browser family and `canvas` is one-way because `canvas` is just an extension of browser family.

Privaricator and our countermeasure aim to change to the browser's true attributes in order to look *different* by making very slight differences at each visit. Utilizing the same method of "little white lies," we

randomize the `canvas` attribute to return a value close to the true value but this returned value is different every time. The `canvas` attribute allows browsers to render graphics on the fly and is typically used in animations or online gaming. Based on the data we gathered to calculate repeatability rates against 6 fingerprinters, we assert that our `canvas` modifications work as often as Privaricator, thus protecting users at the same success rate. The limitation of our countermeasure is its lack of protection against detectability. When we run both our countermeasure and Privaricator against Panopticlick, the most well-known and trusted fingerprinting project, Panopticlick is able to see that we are randomizing canvas but does not see that Privaricator is randomizing `plugins`. (They do not include `offsetHeight` and `offsetWidth` in their fingerprinting algorithm.) We propose a modification to our countermeasure that would protect our countermeasure against detectability. Our experiments only tested 6 fingerprinters, and thus is not an exhaustive or representative set of all types of fingerprinters. There could be some fingerprinters that do not use `plugins` in their algorithms, some that do not use `canvas`, and some that may not use either. For this reason, we argue that combining Privaricator's modification and our modifications into one countermeasure would create an even more robust system that would spoof fingerprinters more than if they were implemented separately. (This is given that the detectability issue with our countermeasure is resolved.) Also, because we are modifying the appearance of `canvas` we also assess the effects on visual appearance of webpages. We find that the most popular websites do not use visible canvases, so our modifications are not seen by users in this cases. But in the instances where canvas is visibly used by websites, there is a noticeable difference between the unmodified Chromium browser and the Chromium browser with our modifications. The specific implementations of Privaricator are explained in Section 2.2 and we expand on our own `canvas` modifications in Section 3.1.1.

2 Background

Many articles outlined in this section describe which specific browser characteristics are used to create fingerprints. As browser fingerprinting has become more popular, there has also been more concern for protection against fingerprinting. We also discuss several countermeasures that attempt to spoof fingerprinters by lying about users' browser environments.

2.1 Fingerprinting in the Wild

Eckersley [3], a study conducted by Electronic Frontier Foundation in 2010, was the first to demonstrate significantly accurate techniques and ease of browser fingerprinting. Users visited the Panopticlick website

and 470,161 fingerprint instances were collected over a year. They found that fingerprints change often (37.4% of users' fingerprints change over time) and that Flash and JavaScript userAgent properties are very accessible, precise ways of identifying unique users. Of the fingerprints that experienced change over time, the study was able to link them to their previous fingerprints with an accuracy rate of 99.1%, showing that fingerprints are easily tracked over time even if a user updates their browser version. In these cases, browser versions were the main difference between a user's old and current fingerprint. They were able to link users to their previous fingerprints because changing to a newer browser version is common and to be expected among users. In this paper, Panoptioclick uses only 8 properties `userAgent`, `HTTP ACCEPT headers`, `cookies enable?`, `screen resolution`, `timezone`, browser plugins, plugin versions and MIME types, system fonts, and partial supercookie test to create a fingerprint.

At its creation, fingerprinting was supposed to be used for good. The motivation for this practice was to be learn a user's behavior so that trackers could notice when there was an anomaly or suspicious behavior in hopes of preventing fraud or other cyber crimes. Also, on some sites, in order to opt-out of a tracking program a user's unique fingerprint must be formed so that the tracker remembers to disregard their data. This study was able to identify 94.2% of users as unique in their sample. This study demonstrates the ease of uniquely identifying users with minimal, and seemingly generic information.

According to Vastel et al. [13] common attributes for finding a user's unique fingerprint include: accept, connection, encoding, headers, languages, userAgent, canvas, cookies, Do Not Track, local storage, platform, plugins, resolution, timezone, WebGL, and fonts. All of these characteristics are categorized as HTTP header, JavaScript, or Flash sourced. This study focused on the ability to track browser fingerprints over time, as browser attributes often change either automatically or manually. They found that they were able to track browsers for 54.48 days and 26% of browsers could be tracked for more than 100 days. 50% of browser instances changed their fingerprints in less than 5 days and 80% changed in less than 10 days. FP-Stalker was able to link fingerprints for a given browser, despite changes, for at least 51 days. It is able to do so based on a rule-based algorithm. This algorithm includes the fact that `OS`, `platform`, and `browser family` must stay consistent, `browser` version either stays constant or increases over time, `local storage`, `Do not Track`, `cookies` and `canvas` settings should stay constant, and allows timezone, resolution, encoding, userAgent, vendor, renderer, plugins, language, accept and headers to change. This study published in 2018 found that attributes that are not expected to change often are canvas (which stays stable for 290 days in 50% of browser instances) and screen resolution (which never changes for 50% of users). These results show that it is relatively easy for fingreprinters to continue tracking users even after attributes change so often.

Naturally occurring changes to a user's browser, like updating to a newer browser version, is not

enough to deter fingerprinting. Thus, fingerprinting is persistent over time and difficult to curb without users' mindful interference. Since this study represents many modern fingerprinter methods, this would be a good area to focus my project on as modifications to these areas are not widely known of or practiced yet. They used a rule-based algorithm to observe how easily users fingerprints could be linked to their own previous fingerprints. Defined in this algorithm, they expect browser family and operating system to be constant and browser version to either be constant or increase over time.

Another previous study by Laperdrix et al. [4] focused on creating their own browser fingerprinting script made up of 17 attributes. These attributes are representative of what modern browser fingerprinters use in the wild. They found that canvas fingerprinting was one of the most discriminating attributes, meaning that it is significant in identifying users since one's canvas is very unlikely to be similar to others'. The HTML `canvas` attribute is used to draw graphics on the fly, via Javascript. With it, paths, shapes, text and images can be drawn, which is useful for dynamic graphics, online gaming, animations, and interactive videos. This means that it allows users to draw graphics on a webpage in real time. `canvas` gives away information on a user's hardware and software environment, for both desktop and mobile browsers. Figure 1 shows what a rendered image of a canvas fingerprint looks like. Because varying browsers use different fonts and emojis and render colors differently, the returned image of the `canvas` can be very useful in identifying users when combined with other attributes to create a full fingerprint. Acar et al. [2] found that canvas fingerprinting is prevalent in 5% of the top 100,000 Alexa sites, which is a high percentage relative to other fingerprinting techniques. They also found some uses of canvas fingerprinting in the wild that was even unreported in the academic research community. Canvas fingerprinting is a technique that renders 2D images, invisible to the user, in order to get information about the browser's image processing in regards to fonts, shapes, colors, and emojis. Fingerprinters rely on the canvas attribute to ensure that information of the user's operating system from the user agent and canvas are consistent. Canvas is used in a way to double-check that a user is not lying. Canvas settings are also reliable because they are assumed to stay relatively stable, or unchanging over time, because it is unlikely that a user will change their configurations for image processing. Vastel et al. [13] found that canvas fingerprints usually stay stable for at least 290 days. There are multiple steps involved in canvas fingerprinting.

1. *Font probing* The script tells the browser to render a string of letters twice. For the first string, it tells the browser to render the string using a fake font name, forcing the browser to use a "fallback" font, or a font that is available to the browser. This "fallback" font will vary depending on the OS and the fonts installed on the system. For the second line, the script tells the browser to render the string using the font Arial, which commonly installed on many operating systems. The string rendered with Arial

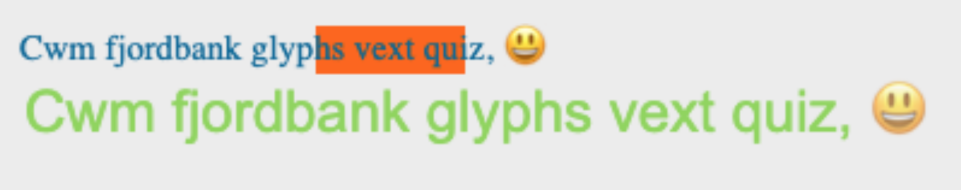

Figure 1: Above is an example of a rendered canvas used in canvas fingerprinting. Several lines of text are rendered in different fonts, colors and transparencies and usually include at least 1 emoji. Shapes are also drawn in different colors and transparencies.

font will have small visible variations of pixels due to differences in rendering processes in operating system.

2. *Emojis* Emojis are small icons or characters that are used to represent a user's emotion. Different operating systems will render the same emoji differently and thus reveal the user's hardware.

Nikiforakis et al. [8] notes that lying about having older browser versions (as seen in use in the Tor browser) is often ineffective because newer version specific capabilities are revealed when scripts are run. Fingerprinting scripts can also detect lies using canvas fingerprinting if the browser's user agent and rendering of an emoji is inconsistent. Lying must be done in a careful manner because fingerprinters often check if a user has employed any kind of fingerprinting prevention tactic. When a fingerprinter notices that a user is lying about their browser characteristics, that user is even more unique in the dataset of all users since fingerprinting awareness and information privacy is not widely known by the average user. In Acar et al. [2], which originally proposed the idea of canvas fingerprinting, used functions `fillText` and `toDataUrl` to draw text on the canvas and read the image's data. Once the image is rendered, the `toDataUrl` method returns the canvas's data in *dataUrl* format, which is an base64 encoded representation of the binary data of the pixels. Then, the fingerprinting script takes a hash of this binary information and uses it as the user's canvas fingerprint which then gets combined with information from other attributes like `plugins`, `platform`, `timezone`, and `language` to form a user's complete browser fingerprint.

2.2 Prevention Methods in Use

Vastel et al. [11] notes that several prevention methods have been created in order to deceive fingerprinters, such as Ultimate User Agent Switcher and User-Agent Switcher. These countermeasures work by changing the information of the user's `userAgent` which include data on name, version and platform of

the browser. The navigator object contains the userAgent but also contains repetitive information such as platform in addition to plugins, cookiesEnabled, and userLanguage. In an unmodified browser, these two attributes' values should be identical. Ultimate User Agent Switcher alters the userAgent sent in the HTTP request but doesn't alter the one in the navigator. Both Ultimate User Agent Switcher and User-Agent Switcher reveal inconsistencies between the userAgent and navigator. The problem with these techniques is that their lying methods for certain attributes are not robust enough and end up revealing the ground truth of the user's environment when other methods that should return the same values are called by the fingerprinter. Fingerprinters use this double-checking technique to discover the unmodified attributes and if the user has employed any prevention methods in order to create a unique fingerprint. Inconsistencies in returned values is the main giveaway spotted by fingerprinters and the reason why many fingerprinting countermeasures ultimately fail.

This is a main issue with the offensive and defensive side of fingerprinting; they are trapped in an arms race. When the defensive side creates a new countermeasure, the fingerprinters will learn from these techniques and incorporate more resilient tests. This study outlines and creates tests for developers to check whether their prevention techniques successfully deceive fingerprinters. They check short-term and long-term fingerprintability. The short-term fingerprintability tests, implemented under the name FP-Scanner [12], check how a browser with a countermeasure differs from a browser without one. It checks for DOM modifications (or changes in HTML), inconsistencies between HTTP headers and the `navigator` object, HTTP header modifications, overridden functions/attributes by checking their `toString` representations, and canvas fingerprinting. Long-term fingerprintability tests focus on how countermeasures affect the ability for a user to be tracked over time. They do this by detecting variability, evaluating long-term tracking by applying the countermeasure to all fingerprints in their test population to link users to their fingerprint before they introduced the countermeasure, and applying browser extension changes on fingerprints. The paper states that FP-TESTER contains four components: a fingerprinter, a short-term test component, a long-term test component, and a report. Upon reaching out the the author of FP-TESTER, it was made clear that the long-term fingerprintability tests had not been implemented but were outlined in the paper as proof-of-concept and motivation to be implemented in the future. In this study, they were able to identify two canvas extensions but failed to identify two user-agent spoofers. We were able to get in contact with one of the author's of FP-TESTER and he directed us to a Github repository of FP-Scanner which is the implementation of the short-term fingerprintability tests outlined in FP-TESTER. According to Github, FP-Scanner was last modified 10 months ago (in May of 2018) so we assume it was developed with modern fingerprinting and countermeasure tactics taken into account. Upon looking at the source code of FP-Scanner further, we see that it is not able to be used against our countermeasure or Privaricator as it is

implemented but instead requires the development of our own database of very specific structure to match theirs, an automated script which can repeatedly scrape data from fingerprinting webpages and populate this database, and changes in FP-Scanner's implementation. Due to all of the components necessary to get FP-Scanner to work, we decide not to use their detectability tests for our study, although if these components were implemented FP-Scanner would have been useful. We discuss the difficulties of implementing FP-Scanner in Section 4.3.

Mulazzani et al. [5] focused on identifying users' web browsers based on underlying JavaScript engines. Using a test set of browsers and browser versions and a decision tree, they were able to ID a user's browser in multiple rounds. They found that modified userAgents were able to be detected. Comparing these two studies, we can deduce that userAgent spoofing techniques are not reliable or consistent prevention or fingerprinting techniques in modern technology. Our countermeasure will focus on modifying canvas and not userAgent, therefore FP-TESTER's short-term fingerprintability tests will still be useful since they have proved to detect other canvas countermeasures.

Privaricator modified Chromium browser using randomization in offsetHeight, offsetWidth, getBoundingClientRect, and plugins as a way to make users look unique every time they are fingerprinted [7]. Their countermeasure, Privaricator, is proposed as a comprehensive approach to preventing fingerprinting. Privaricator's purpose is to make the user appear to have a different fingerprint at every website visit. So, the user is still able to have their fingerprint collected, but the randomization modifications make the user look different from their past and future fingerprints, thereby unlinking their browser history. Browser fingerprinting is therefore rendered useless, as its whole purpose is to keep a record of a user's browsing history over time. Privaricator's policy includes returning 0, a random number between 0-100, and the original value +/- 5% noise to `offsetHeight, offsetWidth, and getBoundingClientRect` respectively. These policies are controlled by a lying threshold (how fast Privaricator starts lying, or in other words after a certain amount of accesses to these values) and a lying probability (the probability or frequency of lying after the threshold has been met). For plugins, they use three parameters to decide the how Privaricator will lie about the browser's plugins. They define probability *P(plug hide)* as the a probability for hiding each individual plugin in a browser's list of plugins, *theta* as how soon Privaricator will start lying, and *P(lie)* as how often Privaricator lies. This technique does not prevent fingerprintability, but instead prevents linkability between fingerprints virtually making fingerprinting useless. Their techniques are helpful in understanding how fingerprints are collected. Using several benchmark suites, they showed that Privaricator caused negligible overhead for the user. By analyzing screenshots of vanilla browsers against screenshots of their modified browser, they also show that there was no detrimental or significant change in appearance of the sites visited by the user.

Fingerprinter Used	Repeatabiity Rate
BlueCava	3.68%
Coinbase	21.64%
Fingerprint.js	21.64%
PetPortal	62.17%

Figure 2: Repeatability rates found by the Privaricator study.

Figure 2 outlines the fingerprinters that the Privaricator study used and their resulting repeatability rates. In this study, they ran automated tests using Privaricator against each fingerprinter at 1,331 different parameters (varying each of the probabilities at intervals of 10%). So for example, PetPortal yielded a repeatability rate of 62.17%, meaning that out of all of their runs, 62.17% of those runs returned fingerprint that was already found by the fingerprinter. The other 37.83% of runs returned fingerprints that were not already found by the fingerprinter. PetPortal was the most effective against Privaricator and it utilized offsetHeight and offsetWidth but not plugin information. Since this paper has been published, several studies have incorporated randomization for these specific attributes as areas to focus on for fingerprinters, aware that prevention frameworks have employed techniques from or like Privaricator [11], [13]. Also, `offsetHeight` is only accessed in 1.87% of fingerprinting scripts, [7]. Therefore, we think that these results could be improved upon by modifying alternative features to protect against fingerprinting more thoroughly, especially attributes that more fingerprinters access.

3 Method

In this section we outline the process of implementing Privaricator and modified `canvas`. We also introduce the methods used to test both countermeasure in order to compare their success and effectiveness. The results from these tests are discussed further in the Results section.

3.1 Implementing Privaricator

We have obtained the patch code from one of the authors of Privaricator and implemented its changes into the most current version of the Chromium browser. Privaricator was originally implemented on Chromium version 34.0.1768.0 in 2015. Since then, many changes have been made to the source code including changes in syntax, class names, function names, and structure of the code. We spent some time deciding which version we wanted to implement our countermeasure on, but we knew that it had to be the same version we implemented Privaricator on in order to draw legitimate comparisons between the two. We were able to find version 34.0.1768.0, but decided against implementing anything here for several reasons. Firstly, hav-

ing a 4 year old version of Chromium may impact the user's experience, who would be most used to using up-to-date versions of browsers. We are assuming that there are many functionality improvements in the newest version compared to version 34.0.1768.0. Secondly, having such an old browser of Chromium may be a very defining attribute for users. This may make it very easy for fingerprinters to identify users or at least classify users in a smaller percentage of people with that version. Many Chromium users have probably updated their version since 2015. We then moved on to finding the most current version of Chromium. Chromium is not very well documented and source code is found on Google's own adapted version of Git (with limited functionality and readability). While further investigating the source code provided by Google, it became apparent that the "Blink" library was not included in it. Blink is a fork of WebCore which is a layout, rendering and Document Object Model library that can be used in Chrome or Opera browsers. It is where all of the Privaricator modifications reside thus a necessary part of the source code that we needed. We were then able to download Chromium with the Blink library included which overall contains over 649,000 files totalling 26.3 GB. Being such a large file, download and compilation time can be quite lengthy.

Once we had the source code ready and confirmed that it compiled, we implemented the Privaricator patch code and confirmed that it also compiles. The Privaricator patch modifies the `Script Controller`, `Navigator`, `Navigator ID`, `Element`, and `Plugin Data` classes. The first three classes contain modifications to the `offsetHeight` and `offsetWidth` attributes and the last two modify the `plugin` information. According to the patch code of Privaricator, `offsetHeight` and `offsetWidth` are randomly changed to +/- 5% of their true values. `plugins` are changed based on a set of rules dictated by the `P(plug hide)`, `offset threshold`, and `P(lie)` global variables. According to one of the authors of Privaricator, these variables are meant to be set by the user as an environment variable on their system and changed depending on the type of fingerprinting script on the website being visited and how many benign websites (websites not containing fingerprinting scripts) are visited. We believe that this implementation of dynamically changing environment variables was most useful to the researchers developing Privaricator in order to run scripted automated tests to attempt the many parameters to find the best combination. We slightly modified the original code of Privaricator to set `P(lie)` to 20, `offset threshold` to 60, and `P(plug hide)` to 50. These values were chosen based off of a chart included in the Results section of the Privaricator paper that highlighted the range of values that were found to be successful. The author of the paper also confirmed that within this range, there is no right or wrong answer to set the values to [6].

3.1.1 Implementing Randomization in canvas

To find where in the source code we have to randomize `canvas`, we visit a few known fingerprinting websites and set a breakpoint in the JavaScript code to pause at any code that accessing `canvas`. This reveals the exact functions used by the fingerprinter to create, modify, and output a `canvas` for canvas fingerprinting. The code for canvas fingerprinting has many similarities between fingerprinters and is relatively simple. First a canvas is created using the desired height and width. A rectangle is then drawn at a given x and y position of a certain height, width and color. Then a few lines of text (usually including emojis) are drawn to the canvas, specified by the type of font, x and y positions, and color. Finally, the canvas is returned to the fingerprinter using the function `toDataURL`. `toDataURL` records the information of each pixel, hashes it using Base64 encoding and returns the resulting string. This allows the image to be converted between an image of the canvas and its string representation. Users are not able to see what the rendered canvas looks like since fingerprinters make it invisible to not reveal that they are fingerprinting visitors. But we use the `toDataURL` output to recreate the canvas and study the effects of our modifications.

The first function we attempt to randomize is the `fillRect` function which creates a rectangle on the canvas. We were unable to successfully randomize the rendered rectangle; the modifications we were making to color did not seem to have any actual effect on the resulting canvas based on the recreated image using `toDataURL`. We randomly selected a color to be filled by the rectangle and we were able to see these changes because the background of every website were populated with different colors. For example, Google's white background appears completely orange at one visit and green at another. Yet, this modification does not change the fingerprint at all. Some of the features of `canvas` are documented online to be used by web developers. For example, we can find documentation that tells us to use the function `fillRect` to draw a rectangle on the canvas. But there is little to no documentation of how to modify the inner workings of Chromium. `fillRect` is actually a wrapper function which calls many other functions within it to complete its function. Attempting to change color within each of the functions within `fillRect`, we still saw that the background of all webpages was being affected. This leads us to believe that the same `fillRect` function is used not only to draw on the canvas, but for any document on the webpage so it may be the case that any randomization added here could affect the appearance of all webpages. So we decide that even if we were able to properly implement randomization to `fillRect` so that the canvas fingerprint changed, the changes in background of every webpage would negatively impact user experience.

Due to time constraints, we shift our focus to other functions that have simpler implementations like those that randomize the color of the text. In the function `fillStyle`, we change the R (red), G (green),

R	$2(.15 * 255) =$	76.5
G	$2(.15 * 255) =$	76.5
B	$2(.15 * 255) =$	76.5
A	$2(.10 * 255) =$	51

Figure 3: Number of possible different values for each text color property in `canvas`

B (blue), and A (alpha which represents transparency) to any value in the range of 0 to 255. This range includes all of the possible values that are valid for these properties and must be integers. This randomization changes the `toDataURL` output each time and fingerprinters return different fingerprints each time. The changes are also visible when we render the image of the canvas. But, this modification changes colors to be completely different than their original value and would not be usable in a practical setting for users. It provides us with a proof of concept though, showing us that randomizing canvas does in fact change our fingerprint. The R, G, B, and A properties have a range of 0 to 255. So, we tune the randomizations to randomly change the R, G, B, and A variables to +/- 10% of the total range closest to the true value (the values specified by the fingerprinter). Since the range has 255 possible values, 10% of this range is 25.5 values below and 25.5 values above the true value for a total of 51 possible values. If the true value of the R value was 50, our modifications randomly choose a value between $(50 - 25.5) = 24.5$ and $(50 + 25.5) = 75.5$. The R, G, B, and A properties require integer values, so we round down to 25 so that the total range of possible values is 50, an even number. This allows us to have the same range of values to randomly select above and below the true value. This still changes the canvas fingerprint and overall fingerprint and the changes are less noticeable in the rendered image of the canvas. But, these significantly impact the number of possible fingerprints we are be able to achieve. The range of +/- 10% allows for R, G, B, and A to each only have up to 50 possible different values. So we tune the randomization to allow R, G, B to randomly assign +/- 15% of the total range and keep A to +/- 10%. The resulting image of our modifications are seen in Figure 4. It follows that 15% of 255 equals 38.25 (a decimal) and doubling this range produces 76.5 (also a decimal). For the same reason, we round the 15% range for the R, G and B properties down to 38 so the full range including values above and below the true value is balanced. We do not believe that the decision to round up or down for these decimal numbers has a significant effect on our countermeasure. This gives R, G, and B each a range of 76 different values and A 50 values (Figure 3).

We also make sure that if the new value is below zero, that variable is set to zero and if it is above 255, it is set to 255. This ensures that no value is out of range and gives an erroneous value that does not make sense to canvas specifications.

To do so, we modify the fillText function which determines the x and y coordinates of the text on the canvas. We randomize the x and y values to be +/- 5 their true values. We also make sure that the x and y

Figure 4: On the left side of the image above is a portion of a canvas fingerprint rendered from an unmodified browser. On the right side is a portion of a canvas fingerprint rendered from a browser with just our color randomizations implemented. This highlights the visible difference in output caused by our randomizations in the R, G, B and A properties.

values are no lower than 0 to stay within range. Adding to the x and y values should not produce a value out of range on the upper bound of the canvas because these values indicate where the text starts. So, x and y values of text are typically low values so that all of the text is within the canvas. Adding to these values will likely keep the text within the canvas's width or height range. We chose the value of 5 because the canvases used in canvas fingerprinting are already small.

3.1.2 Testing Against Fingerprinters

Running preliminary tests against our canvas modifications to color and to position of text separately from each other shows that both modifications are able to change the `toDataURL` output and our overall fingerprints. Once Privaricator and the canvas modifications were fully implemented, we were able to gather large datasets of fingerprints to calculate repeatability rates. Repeatability rates measure the success of a countermeasure on a numeric scale. It is calculated by dividing the number of repeated fingerprints by the total number of fingerprints in the dataset.

$$\text{Repeatability Rate} = \frac{\text{number of repeated fingerprints}}{\text{number of total fingerprints}} \tag{1}$$

For example if 10 fingerprint tests are run and there are 9 distinct fingerprints with the remaining 1 fingerprint being a repeated fingerprint that was previously found, the repeatability rate is 10%. In theory, the lower the repeatability rate, the more effective the countermeasure because the objective is to generate a different fingerprint at each website visit. We test Privaricator and our countermeasure against 6 different fingerprinters: Fingerprintjs2, Hidester, BlueCava, Browserprint, PetPortal, and Fingerprint.js. All fingerprinters are free to use online and return a string indicating the user's fingerprint along with some of the attributes that are included in the fingerprint. (One caveat with this approach is that because these

fingerprinters are free to use online, they may not represent the most rigorous or advanced browser fingerprinting scripts that exist in the wild. Because only we only use 6 fingerprinters, they do not fully represent the broad scope of all existing fingerprinting techniques. Thus we can only discuss our results relative to these fingerprinters and cannot extrapolate this data to the total population of fingerprinters in the wild. Due to lack of financial resources, we do not use any commercial fingerprinters in testing. The only exceptions to this are BlueCava and Fingerprint.js which are a commercial fingerprinter but their homepage returns a fingerprint to the user but they do not reveal which attributes their fingerprinting scripts check for. Their sold fingerprinting services may be more robust.) Another point to note is that Fingerprint.js is the commercial version of Fingerprintjs2. Fingerprint.js is marketed as the "modern and flexible" version of Fingerprintjs2, implying that Fingerprint.js utilizes more advanced fingerprinting technology. So, we include both fingerprinters in our tests to see if there is a noticeable difference in their repeatability rates. We run each fingerprinter a couple hundred times for all countermeasures and record the returned fingerprint. These tests are run manually, by visiting the fingerprinter, navigating to another website, returning to the fingerprinter and repeating this process consecutively. The results of these tests are explained in Section 5.1.

3.2 Assessing Visual Breakage

Because our modifications to `canvas` affect color and position of text, it is important to assess their effects on the appearance of webpages. We visit many of the Alexa top 500 websites, which is a list that ranks websites that are most frequently visited by users of the Alexa toolbar across the globe. In all of the websites we visit, we record screenshots of the website from an unmodified Chromium browser and a Chromium browser containing our modifications. We also visit websites that are known to have visible canvases on them to note the difference between screenshots. To assess all possible user experiences, we include instances where users do need to use canvas. We compare their differences based on our own subjectivity. Conducting a formal user study in which participants are randomly assigned a screenshot from an unmodified or modified browser and then asked to describe anything unusual they see could have been conducted and useful. But we choose not to do this due to a lack of time and resources (lack of compensation for participants) for this study.

3.3 Measuring Detectability

As previously mentioned, it is imperative that fingerprinters are not aware that they are being lied to. Because very few in the general population know about fingerprinting and even fewer people have coun-

termeasures in place, if a fingerprinter can detect that a user is lying it makes the user look even more unique. We were able to test the detectability of Privaricator and our countermeasure by running the countermeasures against Panopticlick. Their test runs online for free and outputs the values of all attributes they access. We trust that Panopticlick is a reliable, robust fingerprinter as this project has been and continues to be widely cited in research. They have also improved upon their original fingerprinting script; they are now on version 3.0. FP-Scanner is an open source test suite written in Python that measures detectability based on a predetermined set of rules. We initally hoped to use it in our experiments to test detectability but implementing it turns out to be a complex process. The current implementation of FP-Scanner accesses an online database set up by USENIX Association, or the Advanced Computing Systems Association, who funded the FP-Scanner project. This database holds information on the true values of an unmodified browser's configuration and the values after a countermeasure is used. It also specifies which countermeasure was used to generate these values. FP-Scanner then takes these values and compares them in their "inconsistency tests" and reports whether they were able to detect changes. In order to use FP-Scanner we must change their implementation to access our own database instead of theirs. Our database must be structured exactly the same as theirs, with corresponding columns to ensure the tests work properly. We would have to gather a large amount of data for true values and spoofed values and enter them into the database. An automated script would be necessary to do this task efficiently. So, we conclude that modifying FP-Scanner and implementing the database and script necessary to use it is not feasible given the time frame of our project. Instead, we only use Panopticlick's online, ready-to-use fingerprinter but do acknowledge that FP-Scanner is a useful tool for the future.

4 Results

4.1 Repeatability Rates

We run Fingerprintjs2, Hidester, BlueCava, Browserprint, PetPortal, and Fingerprint.js fingerprinters against both countermeasures hundreds of times each and record the fingerprint of each run to calculate 12 repeatability rates, seen in Figure 5.

Fingerprintjs2, Hidester, Browserprint, and Fingerprint.js all yield a different fingerprint every time and no fingerprints repeat. Privaricator and our countermeasure yield a repeatability rate of 100% against BlueCava and PetPortal meaning that the same fingerprint is returned every time and the countermeasures do not seem to spoof fingerprinters at all. Repeatability rates are dependent on the fingerprinter used because different scripts use various attributes to create users' fingerprints. Because Fingerprintjs2 is an open

Fingerprinter	Privaricator Repeatabiity Rate	Canvas Repeatabiity Rate	# Trials (Each)
Fingerprintjs2	0%	0%	500
Hidester	0%	0%	500
Browserprint	0%	0%	200
Fingerprint.js	0%	0%	100
BlueCava	100%	100%	200
PetPortal	100%	100%	150

Figure 5: Repeatability rates for Privaricator and our countermeasure found using manual testing.

source fingerprinter, with its code available on Github, we do know that they do not use `offsetHeight` or `offsetWidth`. But their script does include taking the value of `plugins` and `canvas` once, concatenating them with the values of the other attributes gathered, then hashing this long string to create a fingerprint. In this case, both countermeasures effectively change the fingerprint every time. Because the Fingerprintjs2 uses such a simple approach for creating a fingerprint, it is easy to change the whole fingerprint with just one change in any of the attributes. Fingerprintjs2 does check if the user is lying about their operating system, browser family, language, and screen resolution but does not check for lying in the other attributes. Fingerprintjs2 is easily susceptible to small changes, making the randomization in Privaricator and our countermeasure effective. We do not have access to the code for fingerprinting scripts used in Hidester, Browserprint, or Fingerprint.js but we have reason to believe they work in similar ways to Fingerprintjs2 due to their repeatability rates of 0%. These fingeprinters represent very simple approaches to fingerprinting which can easily be broken. Also, Fingerprintjs2 and Fingerprint.js do not behave differently as we suspected telling us that Fingerprint.js's fingerprinting technology does not appear to be significantly more advanced than Fingerprintjs2's. Since they are different versions of almost the same code, any additional features in Fingerprint.js (if they exist) still do not work against our countermeasures.

For our countermeasure, we can calculate the number of possible fingerprints that can be returned to the fingerprinter. Based on our modifications R, G, and B have 76 possible values, A has 50 possible values, and x and y each have 10 possible values we get $(76 * 76 * 76 * 50 * 10 * 10) = 2,184,880,000$ different combinations of values and resulting fingerprints. So, we would only expect to repeat a fingerprint after about this many runs. Similarly, Privaricator has billions of possible combinations of plugins so it would also take a significant amount of runs to repeat a fingerprint. Due to the number of possible fingerprints Privaricator and our countermeasure can theoretically create, it makes sense that we would not repeat a fingerprint after only a couple of hundred of runs especially since their fingerprinting algorithms are so simple. Since we are testing fingerprints manually, we are only able to collect hundreds of fingerprints for each fingerprinter this explains the repeatability rates of 0%. It would be useful to develop an automated script that could run repeated tests on fingerprinters continuously and record the returned fingerprints as

well as a program that would calculate repeatablility rates for very large datasets. Due to time constraints we are unable to create such programs, but we realize their benefits in this study.

When we test BlueCava and PetPortal we obtain repeatability rates of 100%, meaning that they return the same fingerprint every time showing that the randomizations do not have any effect on fingerprintability. We do not know which attributes are used in the BlueCava fingerprinter, but PetPortal does list the attributes they use in theirs. Among the attributes PetPortal lists, `offsetHeight`, `offsetWidth`, `plugins`, and `canvas` are not included. This explains why our fingerprint does not change and we receive a repeatability rate of 100%; the randomizations are not being used at all. We assume that BlueCava's fingerprinter works similarly and does not include any of the attributes modified by Privaricator or our countermeasure or that they implement a clever workaround to randomizations. It could also be the case that BlueCava and PetPortal utilize cookies to recognize repeated visitors. While running our fingerprinting tests, we do not disable cookies so it is possible websites were storing them on our device. If BlueCava and PetPortal store cookies then they would not need to generate new fingerprints each time but would be able to recognize us immediately and return our original fingerprint. The 100% repeatability rates highlight the wide spectrum of existing fingerprinting scripts. As mentioned before, some fingerprinters only use attributes very simply by combining them all into one string and thus are easy to break. But some fingerprinters like BlueCava and PetPortal may not include the attributes Privaricator or our countermeasure modify. Or they may interpret the values in a clever way, like getting values multiple times to check for randomization.

It is interesting to note that PetPortal, BlueCava and Fingerprint.js have differing repeatability rates in our study and in Privaricator's. There could be several explanation for this discrepancy. One reason could be that Privaricator utilized automated scripts to run repeated tests while we run our test manually. They do not specify how many tests they run for each fingerprinter, but it is possible that they ran billions of trials to achieve higher repeatability rates for Fingerprint.js. Another difference between our testing is that we gather fingerprints by navigating to a fingerprinting page, record the fingerprint, leave the page, and then return to the fingerprinting page to repeat the process. We are also uncertain of how Privaricator repeatedly gathered their fingerprints, as it is not explained in the article. If the method in which fingerprints are gathered differs, then repeatability rates may also consequently differ. Finally, Privaricator performed better against (had a lower repeatbility rate for) BlueCava and PetPortal. This may be caused by an improvement in BlueCava's and PetPortal's fingerprinting scripts. We know PetPortal does not use any of the attributes modified by Privaricator or our countermeasure, but it could be the case that they did use them when Privaricator ran their study in 2015. BlueCava is a commercial fingerprinter that sells its services to websites, so it is also likely that their fingerprinting technology has since improved in the last 4 years. Iteratively improving their products is imperative for broswer fingerprinting, as it is a constant arms race

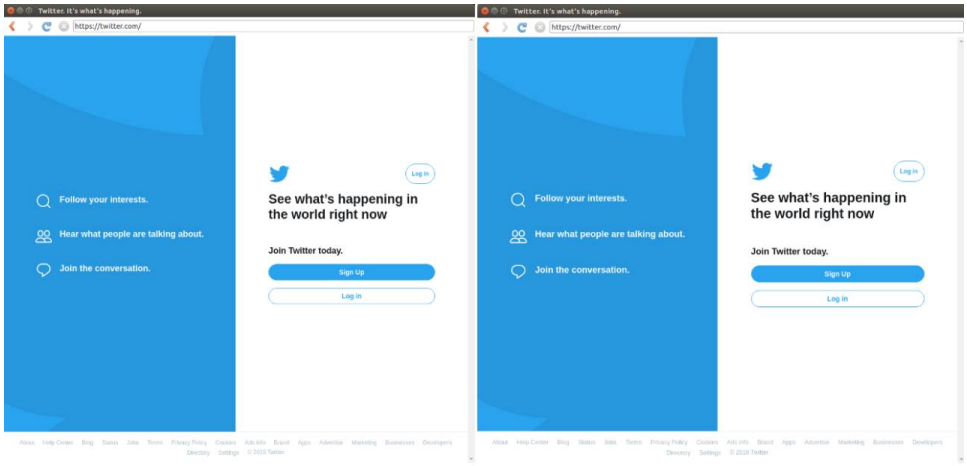

Figure 6: On the left side of the image above is a screenshot of Twitter's homepage taken on an unmodified browser. The right side contains a screenshot of Twitter's homepage taken on a browser with our countermeasure implemented. There is no noticeable difference in their appearances.

of offense (fingerprinters) vs defense (countermeasures).

4.2 Visual Breakage

Our modifications change the color and text of `canvas` so we must be careful to not negatively impact the users' normal online experience. We analyzed the implications of our modifications by assessing screenshots of webpages from an unmodified Chromium browser and a Chromium browser containing our changes. First we compared the differences of popular websites. Some of the websites from the Alexa Top 500 Global sites included Google, Facebook, Amazon, Twitter, Wikipedia, Reddit, Youtube, Baidu, and Qq. Figure 6 illustrates one of these screenshots. Comparing the screenshots of these websites from the two browsers, we were unable to detect any difference caused by our modifications besides normal, expected variation. This variation was due to different ads that populated on the page, suggestions, or changing images and text (typically seen in suggested articles or videos). There was no noticeable difference because popular websites tend to not use visible canvases, thus our modifications are seen only by fingerprinters and not by users.

Next, we purposely visited websites that contained visible canvases. When the screenshots are compared side by side, there is a noticeable difference in color. Figure 7 shows an example of the visible effects of our modifications on canvas. We conclude that these random changes could potentially annoy users who need to use `canvas` functionality. But we argue that looking at the website with the two browsers open side by side would show a noticeable difference yet if the user was only viewing the modified browser, they would not necessarily know that the color is different from its true value. Unless the user frequently visited

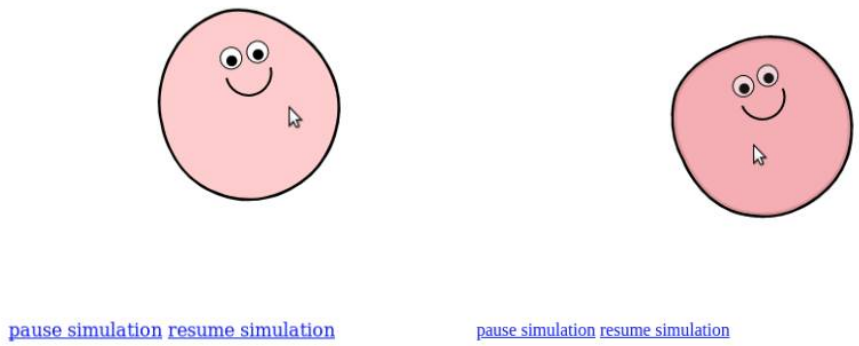

Figure 7: On the left side of the image above is a screenshot of an animation on www.blobsallad.se taken from an unmodified browser. On the right is a screenshot of the same animation taken from a browser with our countermeasure implemented. There is a noticeable difference in color of the figure.

this website, they would have no reason to believe that the color of the canvas was changed. This could irritate users that do visit websites with `canvas` visible often, for example if they play a certain online game frequently or enjoy drawing images online and saving their artwork.

4.3 Detectability

To test for detectability, we visited Panopticlick's website with Privaricator and our countermeasure separately. Panopticlick is the project that spearheads the browser fingerprinting movement by being one of the first to explore fingerprinting techniques in depth and now advertises their own prevention countermeasure. Since they lead much of the browser fingerprinting industry, we visited their online fingerprinter because we assume it has up-to-date and comprehensive tests that check for every kind of known countermeasure technique. Upon visiting the website (50 times each), we found that Privaricator's randomizations specifically in `plugins` does not appear in Panopticlick's test results. The website outputs the attribute that was tested and the value that it received. For Privaricator, different sets of plugin were returned at each run (due to their randomizations), but Panopticlick did not indicate that they were being randomized. This could mean that Panopticlick does not check for randomization in `plugins` or that Privaricator's implementation protects them from being caught. When we ran our countermeasure against Panopticlick, the outputted value for canvas was labeled "randomized" for 48 of the runs and outputted a hash of the canvas for 2 of the runs. The 2 outputted hashes were different from each other, showing randomization worked both of those times. But the 48 runs that returned "randomized" were concerning because this tells us that Panopticlick was able to detect our modifications. Detectability is particularly harmful and defeats the entire purpose of installing a countermeasure. We suspect Panopticlick checks for randomization by calling

rendering the canvas fingerprint several times. We know this because we added a print statement within the `toDataURL` function and this statement was printed several times, but it is also noticeable to users on the website because the page seems to reload several times before the fingerprinting test is complete. We discuss future changes to our countermeasure to prevent detectability in Section 6.3.

5 Future Work

5.1 Minimizing Visual Breakage

A possible way to minimize visual differences in legitimate, non-fingerprinting `canvas` functionality would be to check whether or not the canvas is visible on a webpage or not upon its initialization. If the `canvas` was visible, it would not randomize the color or position values and not impact usability. But if the `canvas` was invisible or hidden, this indicates that it is being used for fingerprinting. In these instances, randomizations should occur to spoof the fingerprinter.

5.2 Avoiding Detectability

Fingerprinters can detect randomization by calling functions repeatedly and comparing the returned results. If the returned values differ every time, this reveals that a countermeasure is randomizing that function. To circumvent this detection, we propose implementing a lie cache. This lie cache would work by storing the first randomized value and returning it for a certain amount of function calls. The amount of times it would return this value would be determined by a threshold, found by comparing the number of accesses are typically used in fingerprinting scripts. Only after this threshold is met (indicating that the current fingerprinting script is finished running), the function would then find a new randomized value and return that the next time it was called (and repeat this process over again). This would protect the countermeasure against detectability because the returned values of the function would match within a fingerprinting run, yet look different across different runs.

5.3 Combining Privaricator and Canvas Randomizations

There are benefits and limitations to using either Privaricator or `canvas` modifications. Privaricator is advantageous because the attributes it randomizes have no direct impact on usability for users. The +/- 5% noise in `offsetHeight` and `offsetWidth` result in changes so small that they are deemed negligible [7]. Randomizing plugins does not change the appearance of webpages nor does it affect the usage of plugins. On the other hand, `offsetHeight` and `offsetWidth` are not widely used in fingerprinting

scripts, making the modifications to these attributes useless in protecting against fingerprinters most of the time. For `canvas` modifications, there is an impact on visual appearance of webpages that could dissatisfy users that need to use legitimate `canvas` functionality. There are also potential fingerprinters that could not include either `plugins` or `canvas` or both in their scripts at all, making both sets of randomizations unhelpful in protecting users. We argue that the ideal fingerprinting countermeasure would be to combine Privaricator and `canvas` randomizations into one countermeasure, with the capability of turning `canvas` randomization off. This would protect users in the case that the fingerprinter does not use either `plugins` or `canvas` and would allow the Privaricator modifications to protect users against fingerprinting if they needed to turn our modifications off in order to use `canvas`. The combination of both countermeasures randomizations would make it extremely unlikely for a user to repeat a fingerprint.

6 Conclusion

Overall, we found that our countermeasure performs the same against most fingerprinters in terms of achieving similar repeatability rates. One significant difference is that Privaricator was not detected, while our countermeasure's randomization was detected against Panopticlick. Detectability is not a trivial issue, though. In order for our countermeasure to be useful it must not be detected by fingerprinters so we acknowledge the importance of implementing a lie cache for `canvas` randomizations. We argue that with a lie cache implemented, our countermeasure would perform as well as Privaricator on average. There are instances where fingerprinters do not use `plugins`, or do not use `canvas`, or do not use either. While canvas modifications can potentially annoy users who need to use the canvas functionality for legitimate purposes, we believe our modifications would not negatively impact general users who visit typical, popular websites. Because our countermeasure was able to spoof some fingerprinters well, we believe that `canvas` modifications are a useful area to focus on when attempting to spoof fingerprinters and that this study highlights some of the possible ways to do so.

References

[1] Gunes Acar et al. "FPDetective: dusting the web for fingerprinters". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 1129–1140.

[2] Gunes Acar et al. "The web never forgets: Persistent tracking mechanisms in the wild". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 674–689.

[3] Peter Eckersley. "How unique is your web browser?" In: *International Symposium on Privacy Enhancing Technologies Symposium*. Springer. 2010, pp. 1–18.

[4] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints". In: *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. 2016, pp. 878–894.

[5] Martin Mulazzani et al. "Fast and reliable browser identification with javascript engine fingerprinting". In: *Web 2.0 Workshop on Security and Privacy (W2SP)*. Vol. 5. Citeseer. 2013.

[6] Nick Nikiforakis. personal communication. May 31, 2019.

[7] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. "Privaricator: Deceiving fingerprinters with little white lies". In: *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2015, pp. 820–830.

[8] Nick Nikiforakis et al. "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting". In: *Security and privacy (SP), 2013 IEEE symposium on*. IEEE. 2013, pp. 541–555.

[9] Joseph Turow et al. "Americans reject tailored advertising and three activities that enable it". In: (2009).

[10] Blase Ur et al. "Smart, useful, scary, creepy: perceptions of online behavioral advertising". In: *proceedings of the eighth symposium on usable privacy and security*. ACM. 2012, p. 4.

[11] Antoine Vastel, Walter Rudametkin, and Romain Rouvoy. "FP-TESTER: Automated Testing of Browser Fingerprint Resilience". In: *IWPE 2018-4th International Workshop on Privacy Engineering*. 2018, pp. 1–5.

[12] Antoine Vastel et al. "FP-scanner: the privacy implications of browser fingerprint inconsistencies". In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 135–150.

[13] Antoine Vastel et al. "FP-STALKER: Tracking Browser Fingerprint Evolutions". In: *IEEE S&P 2018-39th IEEE Symposium on Security and Privacy*. IEEE. 2018, pp. 1–14.