Disguised Chromium Browser: Robust Browser, Flash and Canvas Fingerprinting Protection

Peter Baumann[1], Stefan Katzenbeisser[1], Martin Stopczynski[1], Erik Tews[2]
[1]Technische Universität Darmstadt, Germany and [2]University of Birmingham, UK
peter@vlmc.de, katzenbeisser@seceng.informatik.tu-darmstadt.de,
martin.stopczynski@crisp-da.de, e.tews@bham.ac.uk

ABSTRACT

Browser fingerprinting is a widely used technique to uniquely identify web users and to track their online behavior. Until now, different tools have been proposed to protect the user against browser fingerprinting. However, these tools have usability restrictions as they deactivate browser features and plug-ins (like Flash) or the HTML5 canvas element. In addition, all of them only provide limited protection, as they randomize browser settings with unrealistic parameters or have methodical flaws, making them detectable for trackers.

In this work we demonstrate the first anti-fingerprinting strategy, which protects against Flash fingerprinting without deactivating it, provides robust and undetectable anti-canvas fingerprinting, and uses a large set of real word data to hide the actual system and browser properties without losing usability. We discuss the methods and weaknesses of existing anti-fingerprinting tools in detail and compare them to our enhanced strategies. Our evaluation against real world fingerprinting tools shows a successful fingerprinting protection in over 99% of 70.000 browser sessions.

Keywords

Browser, Fingerprinting, Canvas, Flash, Privacy, Tracking

1. INTRODUCTION

Browser fingerprinting is a technique for uniquely identifying a user through slight variations in the system setup, which are retrievable by websites mostly using JavaScript, Java or Flash scripts [7]. The differences originate from various hardware, network, or software configurations that a user's computer may have [24, 23, 22, 8]. These *fingerprinting features* are widely used to track user behavior across different websites [28] and to establish detailed user profiles [18, 9] for targeted advertisements [29].

Moreover, collected user profiles are often sold between trackers/advertisers and are rarely anonymized by removing real names or addresses [30] (even if user profiles are shared in anonymized form, they can easily be de-anonymized [5]). Connecting this data with information gathered from online social networks, browser location, online shopping or search queries, allows to reveal further privacy sensitive information [26], including personal interests, problems or desires of users, real names and addresses, political or religious views, as well as the financial status. Consequently, detailed user profiles may influence credit scores [19] or risk assessments of health insurers to a disadvantage of the user [6]. Furthermore, the loss of anonymity can lead to price discrimination [39, 21], inference of private data [10] or identify theft/fraud.

Mayer [20] and Eckersley [11] demonstrated that it was possible to uniquely identify 94% of 470,161 users by rather simple fingerprinting techniques. Further studies [1, 2] revealed the prevalence of browser fingerprinting in real world applications and showed that pretty sophisticated fingerprinting methods exist. Fingerprinters commonly use detailed information, such as system fonts, plug-in version numbers, or even differences in graphic rendering engines (using the HTML5 canvas element) as fingerprinting features [23]. Conceptually, every parameter which is rather stable over time and likely different between users, is a potential fingerprinting feature. This also includes exotic features like measuring the dimensions of Unicode glyphs [13], checking for the existence of certain JavaScript functionalities [22, 24], utilizing the drift from exact time in TCP timestamps [17], checking the leakage of the battery status [31] or retrieving a partial subset of visited websites [34]. However, these advanced approaches have limited reliability (e.g., due to network latencies) [15], and are mostly not used in the wild.

To protect the user against fingerprinting, a few tools [32, 27, 12, 38] were proposed, trying to hide unique browser features by randomizing, blocking or deactivating them. Unfortunately, these methods lack usability when browsing websites due to blocked features, they are detectable by the fingerprinter [28, 25, 34, 11] due to unrealistic features, or have flaws making users still identifiable. We expose these issues in Section 2.

In this work, we present a robust approach to protect a user against browser fingerprinting, implemented in our *Disguised Chromium Browser* (DCB). Instead of disabling or randomizing system and browser parameters, we use a large set of real world parameters that are fixed for the entire browsing session and change automatically for the next session. This prevents detectability through unrealistic or constantly changing system parameters, if a fingerprinter requests a parameter multiple times. In addition, DCB is the only tool that protect against Flash as well as canvas fingerprinting without deactivating Flash and HTML5 canvas features. This significantly enhances the usability of web browsers with integrated anti-fingerprinting strategies. Through a thorough study of canvas fingerprinting, we developed a novel and deterministic approach to prevent canvas fingerprinting based on transparently modifying the canvas element in every browsing session. In contrast to other approaches, our solution is not based on adding patterns or noise to the canvas output. Because the canvas element is never unique, the fingerprinter is not able to detect our modifications and is not

WPES'16, October 24 2016, Vienna, Austria

ISBN 978-1-4503-4569-9/16/10. . . $15.00

DOI: http://dx.doi.org/10.1145/2994620.2994621

able to re-identify the user. We show the robustness of the algorithm and demonstrate that every generated canvas element is unique. In contrast to other anti-fingerprinting tools, we also implemented a new protection mechanism against the retrieval of system fonts via Flash.

Finally, we evaluate our solution against real world fingerprinting tools and demonstrate its effective protection against fingerprinting by creating unique fingerprints in over 99% of 70.000 browser sessions. We show that fingerprinters cannot notice the presence of our counter-fingerprinting techniques due to enhanced protection mechanisms inside the browser itself and the usage of real world parameters.

2. RELATED WORK

Various research has been conducted on browser fingerprinting features as well as user protection [8]. To obtain an extensive and up to date list of fingerprinting features, we analyzed popular fingerprinting scripts such as *FingerprintJS*[1], *Coinbase Payment Button*[2], and *BlueCava*[3] as well as data gathered by other researchers [11, 20, 24, 23, 22, 28, 2, 27, 16]. The following list contains popular fingerprinting features that are retrievable through JavaScript, Flash, CSS or the HTTP header [8]:

System information: Device ID, operating system (version, architecture, kernel), screen (resolution, height, width), color depth, pixel depth, timezone, system fonts, system language, date and time, CPU type, clock skew, battery status, mouse movement, keyboard, accelerometer, multitouch capability, microphone, camera, audio capabilities.

Browser information: Browser (version, vendor), User Agent, navigator object, installed plug-ins, preferred and accepted languages, HTTP headers, JavaScript runtime, cookies enabled parameter, supported MIME types, browser history, do-not-track, HTML canvas element, CSS features (font probing, display, etc.).

Network information: IP address, geographic location, TCP timestamps, TCP/IP parameters, proxy-piercing.

Flash Capability Class: Version, manufacturer, serverString, language, screenDPI[4].

Canvas: HTML 5 provides a canvas element that can be used for drawing / rendering 2D graphics and to inspect the image data with pixel accuracy via JavaScript. In order to use the canvas element as fingerprint, typically a fingerprinter first renders a defined text using the function *fillText()* or *strokeText()*. Subsequently, the fingerprinter inspects the unique rendering output using the function *getImageData()*, which returns the RGBA values for every pixel. Similarly, by using the function *toDataURL()*, a Base64-encoded PNG image containing the entire contents of the canvas can be obtained. The unique fingerprint is then produced by hashing the extracted pixel data. Because different graphic cards and rendering engines produce slightly different (but stable) outputs, fingerprinters routinely make use of this element [34, 1].

In the following we discuss prominent anti-fingerprinting tools and show their weaknesses which we were able to overcome within our proposed *Disguised Chromium Browser* (DCB).

The **Tor Browser** [32] is a modification of Firefox for browsing the web through the anonymity network Tor. It implements several countermeasures against browser fingerprinting [25], while focusing on anonymity rather than usability. The anti-fingerprinting techniques rely on disabling specific browser features like plug-ins and the canvas element entirely, resulting in a limited usability and web experience (while those features can still be activated by the user to display common web content like Flash, the protection methods become ineffective). Note that even with activated fingerprinting protection the usage of Tor itself is detectable by fingerprinters [28, 25].

In addition, our experiments confirm that Tor's font probing fingerprint protection, which limits the number of fonts that can be used by the website, can still be circumvented [14] by using many dynamically generated iframes. In summary, the Tor Browser can not effectively protect the user against fingerprinting. In contrast to Tor's strategy, our solution uses a large set of real world data to substitute original browser features; furthermore we manipulate Flash and canvas outputs, rather than disabling these functionalities altogether.

FireGloves [7] is a Firefox browser extension that disables access to specific JavaScript objects (like *navigator.plugins*) instead of disabling browser plug-ins entirely. However, an empty list of plug-ins might be used as fingerprinting feature, since this is not a common behavior. Moreover, plug-ins can still be instantiated and detected. In comparison, DCB performs a smart manipulation of the plug-ins minor version number in order to guarantee plug-in functionality while preventing fingerprinting. Further, FireGloves allows automatic randomization of specific browser settings. This strategy can be detected by fingerprinters through constant changes to (potentially) unrealistic random settings. Similar to the font probing prevention of the Tor Browser, FireGloves limits the number of fonts retrievable by a website – an approach which can be easily circumvented (see above). In addition, other font fingerprinting methods like Flash and canvas font probing are not covered.

In contrast, DCB manipulates the browser internal font handling itself, instead of limiting the font access. Another drawback of FireGloves is the fact that it is a browser extension, and needs to manipulate JavaScript functions instead of directly accessing browser functionalities. This behavior can be detected and potentially circumvented [28, 27]: A fingerprinter might use *Object:getOwnPropertyDescriptor* to check for manipulated JavaScript objects. To avoid this, we implemented DCB directly in the browser.

PriVaricator [27] is a modified Chromium browser that tries to prevent fingerprinting by randomly changing the browser properties whenever they are queried. For example, PriVaricator returns a random subset of the actual plug-in list by filtering out single entries from the *navigator.plugin* property. Nevertheless, since these plug-ins can still be instantiated, it is possible to detect them. Further, PriVaricator randomly manipulates the properties *offsetHeight*, *offsetWidth*, and the function *getBoundingClientRect()* to prevent font probing. Unfortunately, this implementation is detectable by simply checking for deviations in the output of consecutive requests to the same functions. Additionally, PriVaricator provides no means against the retrieval of system fonts using Flash or the HTML 5 canvas element.

FPGuard [12] is a combination of a browser extension and a customized version of Chromium for detecting and preventing browser fingerprinting at runtime. It uses different hard coded heuristics for detecting browser fingerprinting activities on each website separately. The user can then decide to block or randomize the output of the fingerprintable feature on untrustworthy websites.

Unfortunately, the authors do not go into detail on how the fingerprinting features are randomized. More importantly, as shown in [11, 28], simple randomization will increase a user's identifiability, as unrealistic values will occur. Furthermore, as studies have shown

[1] https://valve.github.io/fingerprintjs

[2] https://www.coinbase.com/docs/merchant_tools/payment_buttons

[3] http://bluecava.com/opt-out/

[4] http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/system/Capabilities.html

[40], users are not able to decide whether a website is trustworthy or not and will tend to trust a website, so that fingerprinting protection will likely be disabled by users. In comparison, our solution automatically randomizes fingerprintable settings with a set of realistic values.

As another feature, FPGuard randomizes HTML5 canvas images by adding slight noise before the image is read out. We assume that the image manipulation done by FPGuard is non-deterministic. As stated in previous research [34], this approach is detectable. Fingerprinters can create two identical canvas objects and check for differences in the generated image data. Our solution uses a deterministic and transparent algorithm to manipulate canvas rendering itself. This makes it undetectable to fingerprinters, since the output will be the same for the entire browsing-session and different in the next.

In order to prevent font enumeration, FPGuard randomly hides fonts once a certain number of fonts has been loaded. This approach is also non-deterministic, and can be detected by a fingerprinter by checking for the existence of a font through several independent requests. Flash based font enumeration is only blocked within FPGuard by disabling Flash. This not only reduces the usability of websites, but the absence of Flash can be used as fingerprintable information [22]. Moreover, if the fingerprinter circumvents the above mentioned hardcoded thresholds by requesting only a few browser features through a set of dynamically generated iframes, FPGuard will not be able to provide any protection. In contrast, our solution is automatically applied to the fingerprintable features without disabling any browser capabilities or loss of usability.

FPBlock [38] is another anti-fingerprinting tool intercepting a set of HTTP / JavaScript requests and modifying or blocking requested potential fingerprinting features. The goal of the tool is only to stop third-party, cross-website fingerprinting by returning the same modified browser features saved for the this website, whenever the site is visited again. The main disadvantage of FPBlock is its detectability, since fingerprinters will notice the changing browser features trough subsequently visited websites. Taking our discussion on canvas fingerprinting in Section 4 into account, the canvas fingerprinting solution employed in FPBlock is also potentially detectable, since it only adds random noise to the canvas element.

CanvasFingerprintBlock [4] is a Google Chrome extension that prevents canvas fingerprinting by modifying the functions *toDataURL()* and *getImageData()* to only return an empty image when called. Here, the fingerprint of a canvas element will always by the same, hiding system-specific quirks. Unfortunately, this approach decreases the usability of common websites, and increases the potential identifiability of a user, since the output is always unique. Our implementation does not manipulate the functions that are used to retrieve the image data from the canvas. We chose a transparent and undetectable randomization of the image itself, which does not impair the user experience.

3. DISGUISED CHROMIUM BROWSER

In order to counter fingerprinting and prohibit re-identification of users, two main strategies can be employed: (1) hide fingerprintable features in order to make all users look the same, such as propagated by Tor, and (2) hide the original features through randomization, like done by FireGloves or PriVaricator.

In our anti-fingerprinting research we identified the advantages of both strategies. Using this knowledge we enhanced the existing strategies by new protection mechanisms, and implemented them in one *Disguised Chromium Browser* (DCB). In this section, we first present the architecture of DCB and subsequently describe our strategies, the implementation, and the operation of DCB.

3.1 Architecture and Implementation

We implemented DCB directly in the Chromium browser version 34.0.1847.131 [35] because of three reasons: First, to avoiding the detection of fingerprinting countermeasures through blocked or intercepted JavaScript CallBacks [28, 27]. Second, to bind all fingerprinting protections in one tool. Third, to provide better performance and browser speed.

We decided against modifying the Tor Browser [36], since it applies certain countermeasures against fingerprinting that might impede with our strategies. To tackle the effect of other tracking mechanisms like cookies [8], we implemented our strategies to use the private browsing mode of Chromium. The required effort to port our implementation to newer versions of Chromium mainly depends on the changes that have been applied in Chromium's source code in the meantime. Compiling our modified Chromium browser takes a few minutes. Once updated and compiled, there is no performance loss during runtime. The source code is available online[5].

DCB relies on a client-server architecture, where a server-side algorithm maintains a database of real world fingerprinting features to enforce a robust browser configuration on the client. On the client side, DCB applies the configuration selected by the server, together with the implemented Flash and canvas fingerprinting protection. DCB allows two strategies for distribution of browser configurations to the client:

N:1 – Many Browsers, One Configuration
Since fingerprinters collect a multitude of fingerprinting features, the likelihood of several browsers sharing the same system properties is considerably low. For that reason, when fingerprinters observe the same fingerprint more than once, they most likely will assume the fingerprint to belong to the same browser as in a previous observation. The $N:1$ strategy thwarts this approach by configuring as many browsers (N) as possible to share the same configuration, making all those browsers look the same to the fingerprinter. This decreases the surprisal of observing a specific configuration and limits the re-identification of one specific user.

1:N – One Browser, Many Configurations
In the $1:N$ strategy, the actual browser and system configuration is hidden to the outer world in order to prevent re-identification by fingerprinters. DCB in $1:N$ mode constantly changes the browser configuration on each start. Compared to existing tools like PriVaricator [27] or FPGuard [12], which randomly change fingerprintable parameters, our approach randomizes features to realistic values and prevents constantly changing system parameters. This limits detectability.

3.2 Configuration Server

Every browser has a configuration (its original), consisting of single-value features (like the Windows version), and of multi-value features (such as lists of fonts). The configuration server is responsible to store such real world system and browser properties (fingerprinting features) and to assign configurations to clients based on the selected strategy. To ensure a realistic randomization of features, we use a pre-stored data set of 23,709 real-world fingerprinting features. These anonymized features were gathered in a large scale study [37] similar to Panopticlick [11]. Furthermore, upon DCB startup, clients send their initial configuration to the server, which is added to the database.

DCB can work either with a local or a global configuration server. To guarantee trust, configuration servers may be operated by trust-

[5]http://www.seceng.informatik.tu-darmstadt.de/index.php/research/software/, 8 GB modified Chromium source to be compiled.

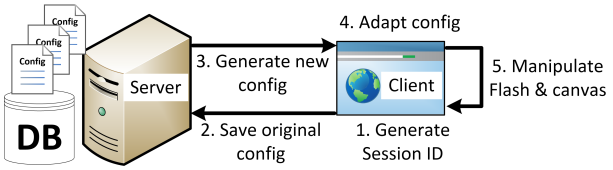

Figure 1: DCB: Initialization of a browser session and mode of operation.

worthy organizations such as the CCC[6] or EFF[7]. Tackling potential single-point-of-failure problems, the amount of servers needs to be raised while the user base grows. To avoid traceability and privacy problems, the server never saves any connection details such as the IP address of users, and can be verified due to the open source implementation.

We implemented the configuration server using CakePHP[8], since it is an easily expandable, model-view-control orientated framework. The upload of a client's configuration is done via HTTPS. Once the client identifies a configuration change, the new configuration will be sent to the sever.

3.3 Configuration Groups

It is important to pay attention to inconsistencies and usability constraints when changing or hiding system and browser properties. For example, if a specific operating system (OS) is propagated, the list of plug-ins reported should match the OS. We decided to use groups of configurations sharing similar values, so that no browser adapts a configuration that contradicts its actual system configuration. One specific configuration group could consist of all users using browsers on the same OS and language. Here, all browsers in this configuration group will only get those configurations (e.g., list of plug-ins), that are available for the given OS. Because a large number of configuration groups is counterproductive in hiding the user effectively, we predefined configuration groups for the user's language and different Windows versions (like Windows version NT6.1 & English or Windows version NT5.0 & French). The fewer groups exist, the smaller the surprisal observing any browser.

Depending on the actual system and therefore the configuration group, the $N:1$ strategy aims to set all fingerprintable features to the most frequent parameters in this configuration group, fitting with the user's environment. This guarantees robustness and usability. At the moment of development, the configuration contained the following parameters: screen information, browser language, user agent (Chrome & WebKit/Blink version, Windows architecture), time and date, system fonts and plug-ins. Further, we use the most common intersection of fonts and plug-ins of all browsers in the group.

3.4 DCB: Mode of Operation

Once a new browser session is started, several steps are performed, shown in Figure 1. First, DCB generates a random session identifier ID (step 1). The ID is used as input in the canvas fingerprinting protection algorithm as well as an identifier for the browsing session. In the second step, DCB collects configuration information on the client (such as fonts, User Agent, or plug-ins) into a JSON encoded configuration file and sends it to the server.

If the $N:1$ strategy is selected, the configuration server checks if there exists a *configuration group* of clients that share similar features. If not, the server creates a new group that fits the browser's configuration. Otherwise, if a browser fits into an existing group, the group's configuration is revised and if necessary updated. This may be necessary to adjust the property of a feature to the majority of users' configurations. Subsequently, the server returns the selected configuration, encoded in JSON, to the DCB browser (step 3). This strategy assures that the generated group configuration is based on the most frequent, already stored configuration data, and the best fitting configuration group for the client (depending on general system features). The browser then adapts the new configuration as its own (step 4), and manipulates the DLL file of Adobe Flash Player (step 5) in order to change propagated values such as the screen resolution or operating system. A full description of the manipulation of Flash and system fonts are presented in Appendix 8.1. Specific manipulations of the fingerprinting features according to the $N:1$ strategy are detailed in Appendix 8.2.

If the $1:N$ strategy is used, the server responds with a random configuration taken from the database. This configuration is encoded in JSON and subsequently sent to the browser (step 3). Steps 4 and 5 are equivalent to the procedure described for $N:1$. Since DCB modifies browser and Flash equally, a fingerprinter will see the same parameters when settings are retrieved by JavaScript or Flash. Finally, changes are fixed for the entire browsing session and reset automatically for the next session. Details on how fingerprinting features are manipulated in the $1:N$ case are given in Appendix 8.3.

Note that a server is needed to gather and generate accurate configurations for both strategies. The server aides in adapting one configuration for a set of users ($1:N$); in addition it stores realistic (original) configurations for the $N:1$ strategy.

4. CANVAS ANTI-FINGERPRINTING

Existing canvas anti-fingerprinting tools manipulate the canvas readout functions *toDataURL()* and *getImageData()* so that random pixel noise (e.g. changing colors) is added in order to prevent fingerprinting. As noted in [34], it does not suffice to add random noise whenever image data is requested from the canvas. Fingerprinters can detect this noise through subsequent identical function calls, comparing the results (image data) and filter changes. Various strategies could be applied to counter the detection of modifying *getImageData()* or *toDataURL()*, but we argue that any approach would face the common problem of detectability (see Appendix 8.4). In contrast, in our approach we modify the rendering of the canvas itself, and always perform the same modification for the entire browsing session.

4.1 Robust Canvas Fingerprinting Protection

Instead of randomly manipulating the canvas readout functions *toDataURL()* or *getImageData()*, DCB deterministically changes the canvas rendering function[9] directly for each browsing session. This function is using *fillText()* and *strokeText()*, which covers all known canvas fingerprinting approaches. Moreover, the approach can be applied to new rendering functions used for fingerprinting.

We use the random session identifier, generated at Chromium startup, to steer the modifications. Due to the randomness of the session identifier, it is guaranteed that *fillText()* returns deterministic values during a browser session but different ones in subsequent browser sessions. Figure 2 illustrates the process.

When the internal method for handling *fillText()* requests is called, we first backup the image buffer (step 1), and wait until the function has finished rendering text (as part of the fingerprinting process) into the buffer (step 2). In the next step we compare the previously

[6]https://www.ccc.de
[7]https://www.eff.org
[8]http://cakephp.org
[9]CanvasRenderingContext2D::drawTextInternal()

Figure 2: Canvas processing algorithm.

saved image data with the new image data in the buffer and store the positions of every pixel that has changed (step 3). In the next step we apply our image manipulation algorithm (step 4) as described in the next paragraph. Finally, the image data is returned (step 5).

4.2 Image manipulation algorithm

The algorithm is implemented in C++, since native functions will render canvas elements faster[10]. The image manipulation algorithm works on a per-pixel basis, and is only applied to pixels that are close to a color border; i.e., to pixels where its top, right, left, and bottom neighbor do not share the same values. We encode a pixel p as (r,g,b,a), where r,b,g,a are the red, green, blue, and alpha values $r,g,b,a \in \{0,..., 255\}$. We modify p to p' by adding an offset to r,g,b. We first concatenate r,g,b and a with the pseudo random session identifier s generated at startup, apply the SHA256 hash, and call the result t. The offset is computed by taking subsequent blocks of 20 characters from t modulo a constant c, which is by default set to 3. Then, for each r,g,b we either add or subtract the offset depending on whether their value was above or below 128, resulting in p'.

Note that the image manipulation is deterministic and due to the slight image changes not visible to the user (using the default setting $c = 3$). Therefore, it is not possible for fingerprinters to detect this strategy and to remove or subtract any modification from the canvas to reconstruct the original image, like suggested in [3]. Using high values for c, for example $c = 50$, is not necessary, as this does not increase the effectiveness of the algorithm and will only make the changes more visible (blurring the image). Assuming a canvas with 1806 pixels, containing a total of 1030 different colors, and using the standard value of $c = 3$, we already get $3^{1030} \approx 2.722 \times 10^{491}$ possible combinations in changing the pixel values. As shown in Section 5.3, this is enough to prevent fingerprinting.

5. EVALUATION

We tested DCB with both strategies $N : 1$ (many browser, one configuration) and $1 : N$ (one browser, many configurations) for its effectiveness against real world fingerprinters. In addition, we also evaluated our canvas anti-fingerprinting strategy independently from the other features to show the robustness and practicality of our approach.

5.1 1:N – One Browser, Many Configurations

The goal of the $1 : N$ strategy is to establish on each browser startup a completely new configuration (fingerprint) so that fingerprinters will not be able to re-identify the user in the next browser session. In addition, the strategy shall guarantee realistic settings and an unchanged user experience on the web.

We examined the effectiveness of DCB running in $1 : N$ mode by analyzing the reported fingerprint of three popular fingerprinters: *FingerprintJS*, *Coinbase Payment Button*, and *BlueCava*.

[10]A JavaScript implementation would be possible. However, JavaScript CallBacks are detectable and can potentially be circumvented. We also noticed a slight delay of 1-2 seconds per page in our first implementation, therefor we favored a native implementation.

5.1.1 Experimental Set-Up

We evaluated DCB under realistic conditions on a standard Windows PC. To automatically simulate user behavior and the generation of a new (modified) configuration by DCB, we started the browser 10,000 times on specific web pages (described next) for each fingerprinter. In every session and for every fingerprinter we saved and compared the generated fingerprint.

FingerprintJS is an open source fingerprinting library written in JavaScript. For retrieving FingerprintJS fingerprints we created a web page that sends the fingerprint to a PHP script via an AJAX request. To derive the fingerprint, we chose three fingerprinting feature sets FS1-FS3 that were present in an exemplary HTML file delivered along with the library. Feature set FS1 includes the features: User Agent, navigator language, color depth, time zone offset, plug-in list, *this.hasSessionStorage()*, *this.hasLocalStorage()*, *window.indexedDB*, *typeof document.body.addBehavior*, *typeof undefined*, *typeof window.openDatabase*, *navigator.cpuClass*, *navigator.platform*, *navigator.doNotTrack*. FS2 uses the same fingerprinting features along with canvas information. FS3 adds the screen resolution to the fingerprint of FS1.

Coinbase generates a fingerprint on the client via JavaScript (its intended use is to protect Bitcoin payments against fraud). We identified two functions, *browserAnalytics()* and *fingerprint()*, that Coinbase uses to retrieve various information[11] about a user's computer. While *browserAnalytics()* returns the information in plaintext, *fingerprint()* creates the actual fingerprint by hashing the collected information using MD5. In order to compare the values of *browserAnalytics()*, we also hash them with MD5 and store them along with the value returned by *fingerprint()* in the database. For this we isolated the fingerprinting code from the Coinbase website and applied it on our own website.

BlueCava is a top 5 fingerprinter [27] that calculates its fingerprint on the server side using a large quantity of fingerprinting features[12]. However, it provides an opt-out page where the fingerprint of the browser can be retrieved. We make use of this feature in our test and grab the fingerprint in an automated manner by using JavaScript.

5.1.2 Fingerprint Robustness of BlueCava

Eckersley [11] suggested that a fingerprinter might easily compensate changes in some of the used features. In order to evaluate the effectiveness of our strategies, we first analyzed the robustness of fingerprints against manual configuration changes. For this experiment we used BlueCava, since it scans a large set of fingerprinting features.

We set up a virtual machine running Windows 7 and installed Google Chrome version 34. We then installed a set of 2000 fonts (BlueCava uses font probing with Flash and JavaScript [28]) and started Chrome in private mode to retrieve the fingerprint from BlueCava's opt-out page. To estimate the robustness to changing fingerprint features, we compared the fingerprints on random system font changes, by adding 5 fonts, then deleting 173 fonts. In all cases the fingerprint was identical. The fingerprint changed after deleting all fonts apart from the standard system fonts and additionally disabling four of Chrome's standard plug-ins (not Flash). This indicates that BlueCava's fingerprint seems to be robust towards changes of a user's system configuration, making it a good benchmark to evaluate the effectiveness of our $1 : N$ strategy.

5.1.3 Results

Table 1 presents the effectiveness of our $1 : N$ strategy in creating

[11]https://coinbase.com/assets/application.js
[12]http://ds.bluecava.com/v50/AC/BCAC5.js

Table 1: Fingerprint results evaluating 1:N strategy in 10,000 browser sessions [higher is better], and N:1 on 12 systems [lower is better].

Fingerprints in strategy	Bluecava	FingerprintJS			Coinbase		Canvas script
		FS1	FS2	FS3	FP1	FP2	
1:N	10,000	9,910	10,000	9,992	10,000	10,000	10,000
N:1	1	1	2	1	7	8	N/A

distinct fingerprints. For both BlueCava and Coinbase, every fingerprint in the set of 10,000 fingerprints of different DCB browser sessions turned out to be unique. As the fingerprint change protection of BlueCava had been outwitted over the course of all 10,000 browser sessions, it would have been impossible for BlueCava to track DCB users over consecutive browser sessions.

On FingerprintJS we recorded 99 duplicates on two out of three tested feature sets. For the features set FS1, 88 fingerprints occurred twice and one fingerprint occurred three times. We explain this behavior by the small amount of features that FingerprintJS collects to generate the fingerprint. FS3, which is an extension of FS1, additionally using the computer's screen resolution, produced only nine duplicates, as it uses more fingerprinting features that can be modified by DCB. In practice, if a fingerprinter uses a smaller fingerprint feature set, the fingerprint becomes less reliable and also less unique, as can be seen in column FS1 and FS3. Note that FingerprintJS collects no information about the system fonts, which normally contributes greatly to a fingerprint's entropy and would otherwise produce a unique fingerprint like for FS2.

Further, FS2, which adds HTML 5 canvas information to FS1's fingerprinted information, resulted in a set of 10,000 unique fingerprints. This is due to our canvas fingerprinting protection mechanism described in Section 4.1. In order to assess the effectives of our strategy against canvas fingerprinting we also applied a separate test run of canvas specific fingerprinting in Section 5.3.

5.1.4 Preliminary Conclusion

The non-commercial open source library FingerprintJS produced over 99% unique fingerprints in 10,000 browser sessions. The commercial fingerprinting scripts of Coinbase and BlueCava produced 100% unique fingerprints in the run of 10,000 browser sessions. More importantly, even though BlueCava is applying means to compensate for fingerprint changes, our $1 : N$ strategy produced changes that were large enough to break out of these compensation mechanisms. Therefore, we can conclude that our $1 : N$ strategy is highly effective against long-term traceability, as a fingerprint cannot be used to identify a user over consecutive browser sessions. In addition, we ensure website usability through automatic Flash and canvas anti-fingerprinting protection mechanisms without disabling those features. We also manually verified the perfect functionality of the top 20 ranked *Alexa.com* websites.

5.2 N:1 – Many Browsers, One Configuration

The $N : 1$ strategy aims at decreasing the surprisal of observing one browser instance by increasing the number of browsers sharing the same configuration. Therefore, evaluating this strategy required the execution of DCB on several systems with varying configurations. As above, we tested the $N : 1$ strategy by matching fingerprints generated by the three introduced fingerprinters.

5.2.1 Experimental Set-Up

We installed our DCB on ten virtual machines (VM) and on two physical computers as a reference. Since we wanted to guarantee that these systems shared the same configuration group, we configured all systems to have the same operating system (Windows 7, Home Premium 32 bit or Professional 64 bit with Service Pack 1), and set the same browser language. We deliberately prepared different VMs, by installing various programs, including different office suites, PDF readers and other software. After the installation we observed that up to 105 additional fonts were installed, with an average of 22 fonts per virtual machine, and up to ten new browser plug-ins. In addition, we added another 25 distinct fonts for every VM, chose different screen resolutions, and selected various time zones. Then, we executed DCB in $N : 1$ mode on all 12 systems and tested it separately against each fingerprinter (BlueCava, Coinbase, FingerprintJS) and the canvas fingerprinting script presented in Section 5.3.

5.2.2 Results

Table 1 summarizes the results. For BlueCava, as well as the feature set FS1 and FS3 on FingerprintJS, all 12 systems generated the same fingerprint, as intended. FS2, which adds the canvas element, generated one identical fingerprint on one of the PCs, since the canvas protection mechanism can not be shared between DCB browsers and is therefore not implemented in $N : 1$[13]. For the Coinbase script our algorithm performed not as good as expected. Out of 12 systems we noticed 7 fingerprint duplicates on FP1 and 8 duplicates on FP2. This could be explained by the fact that Coinbase applies less sophisticated compensation mechanism against configuration changes or only checks for a small fixed set of features, and taking less detailed fingerprints into account.

We can conclude that the $N : 1$ strategy is prone to additional fingerprinting features that might not be covered by the implementation, as additional entropy for distinguishing between otherwise equal configurations might be introduced. Therefore, this requires to constantly consider new fingerprinting features.

5.3 Canvas Fingerprinting Set-Up

We implemented our countermeasures against canvas fingerprinting in conjunction with the $1 : N$ strategy. We isolated the canvas fingerprinting script of AddThis[14] (a popular fingerprinter [1]), implemented it on our website and examined the reported fingerprints in an extensive study. In a run of 10,000 browser sessions we stored every generated fingerprinting canvas image as MD5 hash in our database. To create the rendered image, the canvas fingerprinting function uses *fillText()*, rendering text with the fallback font.

5.3.1 Results

It turned out that every single fingerprinting image in our set of 10,000 browser sessions was unique (Table 1, column Canvas). Therefore, our strategy against canvas fingerprinting is highly effective, as duplicate fingerprints are nearly impossible (see Section 4.1).

[13] Implementing canvas fingerprinting protection in $N : 1$ would require a synchronization of the rendering data using a central rendering engine, which would produce a large overhead in computation and communication time.

[14] http://ct1.addthis.com/static/r07/core130.js

5.4 Comparison of Anti-Fingerprinting Features

Finally, we show in Table 2 the advantages of DCB by comparing it against other anti-fingerprinting tools: Tor Browser version 4.0.8 [36], FireGloves [7], FPGuard [16], FPBlock [38], PriVaricator [27]. We were able to successfully overcome the weaknesses and flaws of existing anti-fingerprinting tools and additionally implemented new Flash as well as canvas fingerprinting protection.

6. CONCLUSION AND FUTURE WORK

In this work we showed flaws in existing anti-fingerprinting tools and presented new approaches preventing fingerprinting. We implemented and evaluated the effectiveness of two strategies $N:1$ and $1:N$, which enhance browser/system fingerprinting protection with the first built-in Flash fingerprinting protection without deactivating Flash. The $N:1$ strategy aims at decreasing the chance of being uniquely identifiable in a large set of DCB browser instances by using the same browser configuration for all users. The $1:N$ strategy aims at breaking the fingerprint by applying major browser/system configuration changes using real word properties every time a new browser session is started.

Furthermore, we demonstrated the effectiveness of the first robust solution against canvas fingerprinting that, in contrast to other approaches, does neither block nor randomly manipulate any canvas functionality used to retrieve the rendered image data.

While our solution is resistant to tracking via cookies by employing the private browsing mode, we have no means to prevent tracking via IP addresses. This could be solved by using an anonymity network (proxy servers). Regarding the manipulation of Flash binaries, we intend to implement our manipulation functionality directly in the browser rather than using an external Python script. Moreover, an automatic configuration update is planned when the user does not restart the session but reconnects to the web. In a future study, we also want to examine whether fingerprint databases of fingerprinters aiming at detecting configuration changes could be poisoned by a large set of false data, caused by an extensive usage of our $1:N$ implementation. For future work we want to implement further functionality which by now are neither implemented in existing tools, nor used by fingerprinters in the wild (mostly because of limited reliability and network latency [15]).

Table 2: Comparison of fingerprinting feature coverage: PriVaricator (PV), FireGloves (FG), FPGuard (FPG), FPBlock (FPB), Tor, DCB

Feature	Retrieval	PV	FG	FPG	FPB	Tor	DCB
Browser History	Rend. timing attack	✓	x	?	x	✓	✓
Browser	JS engine	x	x	x	x	x$_0$	x
	navigator.appVersion	x	✓$_1$	✓$_2$	x	✓$_3$	✓
Clock skew	TCP/ICMP	x	x	x	x	✓$_4$	x
Flash Version	Adope Flash	x	x	✓$_5$	✓$_5$	✓$_5$	✓
	JavaScript	x	✓$_1$	x	?	✓$_3$	✓
Fonts	Using Flash	x	x	✓$_5$	✓$_5$	✓$_5$	✓
	Using CSS	✓$_6$	✓$_6$	✓$_6$	✓$_6$	✓$_6$	✓
	HTML5 canvas	x	x	x	x	✓$_6$	✓
IP/Network info	TCP/IP stack	x	x	x	x	✓	x
Language	HTTP header	x	✓$_1$	?	✓	✓$_3$	✓
	JavaScript	x	✓$_1$	?	✓	✓$_3$	✓
Math constants	JavaScript	x	x	x	x	x	x
Mime types	JavaScript	✓$_7$	✓$_7$	✓$_7$	✓$_7$	✓$_7$	✓
Plugins	JavaScript	✓$_7$	✓$_7$	✓$_7$	✓$_7$	✓$_7$	✓
Canvas	JavaScript	x	x	✓$_8$	✓$_8$	✓$_2$	✓
Screen height	JavaScript	x	✓$_1$	✓$_2$	✓	✓$_3$	✓
Screen width	JavaScript	x	✓$_1$	✓$_2$	✓	✓$_3$	✓
Screen color depth	JavaScript	x	x	✓$_2$	✓	✓$_3$	✓
Screen pixel depth	JavaScript	x	x	✓$_2$	✓	✓$_3$	✓
Screen resolution	Adobe Flash	x	x	✓$_5$	✓$_5$	✓$_5$	✓
	JavaScript	x	✓$_1$	✓$_2$	✓	✓$_3$	✓
Time & date	JavaScript	x	x	x	x	✓$_3$	✓
Time zone offset	JavaScript	x	✓$_1$	x	✓	✓$_3$	✓
User Agent	HTTP header	x	✓$_1$	?	✓	✓$_3$	✓
	JavaScript	x	✓$_1$	✓$_2$	✓	✓$_3$	✓
Windows version	Adobe Flash	x	x	✓$_5$	✓$_5$	x$_5$	✓
	User Agent	x	✓$_1$	✓$_2$	✓	✓$_3$	✓

0) Protection only implemented for a subset of functions. 1) Randomization through constantly changing unrealistic system parameters causes detectability. 2) Expects user approval. Inexperienced users might reveal data to trackers. 3) Blocks access or sets parameter to a fixed value. This causes usability constraints and can be used as fingerprintable information. 4) IP address is hidden due to default usage of an anonymity network. 5) Blocks Flash and impedes the usability of websites. Flash can be activated by the user and become a source of fingerprintable information. 6) Only limits font probing. This can be circumvented by using dynamic generated iframs scanning for a small amount of fonts. 7) Randomly removes plug-ins or adds non existing plug-ins from/to plug-in list (resulting in usability constraints and detectability since plug-ins can still be instantiated). 8) Non-deterministic usage by adding noise limits usability. This makes it detectable by creating multiple canvas outputs in same browsing session and by checking for differences.

7. 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 2014 ACM SIGSAC Conference on Computer & Communications Security*, CCS '14, pages 674–689, New York, NY, USA, 2014. ACM.

[2] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. FPDetective: Dusting the Web for Fingerprinters. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 1129–1140, New York, NY, USA, 2013. ACM.

[3] D. Agrawal and C. C. Aggarwal. On the Design and Quantification of Privacy Preserving Data Mining Algorithms. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, pages 247–255, New York, NY, USA, 2001. ACM.

[4] Appodrome. CanvasFingerprintBlock. https://chrome.google.com/webstore/detail/canvasfingerprintblock/ipmjngkmngdcdpmgmiebdmfbkcecdndc, 2014. Accessed on 2015-02-02.

[5] M. Barbaro and T. Zeller, Jr. A Face Is Exposed for AOL Searcher No. 4417749.

http://www.nytimes.com/2006/08/09/technology/09aol.html?ex=1312776000&en=f6f61949c6da4d38&ei=5090, 2006. Accessed on 2013-10-25.

[6] J. Barnes. Big data bring risks and benefits to insurance customers. http://www.ft.com/cms/s/0/21e289c4-97ef-11e3-8dc3-00144feab7de.html#axzz41oCbtf9J, 2014.

[7] K. Boda, Á. M. Földes, G. G. Gulyás, and S. Imre. User Tracking on the Web via Cross-Browser Fingerprinting. In *Information Security Technology for Applications*, pages 31–46. Springer, 2012.

[8] T. Bujlow, V. Carela-Español, J. Solé-Pareta, and P. Barlet-Ros. Web Tracking: Mechanisms, Implications, and Defenses. *CoRR*, abs/1507.07872, 2015.

[9] C. Castelluccia, M.-A. Kaafar, and M.-D. Tran. Betrayed by Your Ads!: Reconstructing User Profiles from Targeted Ads. In *Proceedings of the 12th International Conference on Privacy Enhancing Technologies*, PETS'12, pages 1–17, Berlin, Heidelberg, 2012. Springer-Verlag.

[10] Duhigg C. How Companies Learn Your Secrets. http://www.nytimes.com/2012/02/19/magazine/shopping-habits.html?pagewanted=all&_r=0, 2012. Accessed on 2016-02-18.

[11] P. Eckersley. How Unique is Your Web Browser? In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, PETS'10, pages 1–18, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] A. FaizKhademi, M. Zulkernine, and K. Weldemariam. FPGuard: Detection and Prevention of Browser Fingerprinting. In P. Samarati, editor, *Data and Applications Security and Privacy XXIX - 29th Annual IFIP WG 11.3 Working Conference, DBSec 2015, Fairfax, VA, USA, July 13-15, 2015, Proceedings*, volume 9149 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2015.

[13] D. Fifield and S. Egelman. Fingerprinting Web Users Through Font Metrics. In R. Böhme and T. Okamoto, editors, *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, volume 8975 of *Lecture Notes in Computer Science*, pages 107–124. Springer, 2015.

[14] Gacar. Circumventing TOR font-limits by using multiple frames. https://www.torproject.org/projects/torbrowser.html.en, 2013.

[15] D. Herrmann, K. Fuchs, and H. Federrath. Fingerprinting Techniques for Target-oriented Investigations in Network Forensics. In S. Katzenbeisser, V. Lotz, and E. R. Weippl, editors, *Sicherheit 2014: Sicherheit, Schutz und Zuverlässigkeit, Beiträge der 7. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), 19.-21. März 2014, Wien, Österreich*, volume 228 of *LNI*, pages 375–390. GI, 2014.

[16] A. Khademi. Browser Fingerprinting: Analysis, Detection, and Prevention at Runtime. In *M.S. thesis, School of Computing, Queen's University, Kingston,*, 2014.

[17] T. Kohno, A. Broido, and K. Claffy. Remote Physical Device Fingerprinting. In *Security and Privacy, 2005 IEEE Symposium on*, pages 211–225, May 2005.

[18] B. Krishnamurthy and C. Wills. Privacy Diffusion on the Web: A Longitudinal Perspective. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 541–550, New York, NY, USA, 2009. ACM.

[19] K. Lobosco. Facebook friends could change your credit score. http://money.cnn.com/2013/08/26/technology/social/facebook-credit-score/index.html, 2013. Accessed on 2013-10-25.

[20] J. Mayer. Any person... a pamphleteer: Internet Anonymity in the Age of Web 2.0. Master's thesis, Undergraduate Senior Thesis, Princeton University, 2009.

[21] J. Mikians, L. Gyarmati, V. Erramilli, and N. Laoutaris. Detecting Price and Search Discrimination on the Internet. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 79–84, New York, NY, USA, 2012. ACM.

[22] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting Information in JavaScript Implementations. In *Proceedings of W2SP 2011*. IEEE Computer Society, may 2011.

[23] K. Mowery and H. Shacham. Pixel Perfect: Fingerprinting Canvas in HTML5. In *Proceedings of W2SP 2012*. IEEE Computer Society, may 2012.

[24] M. Mulazzani and P. Reschl. Fast and reliable browser identification with javascript engine fingerprinting. In *Proceedings of W2SP 2011*. IEEE Computer Society, 2013.

[25] S. Murdoch, M. Perry, and E. Clark. Tor: Cross-origin fingerprinting unlinkabilit. https://www.torproject.org/projects/torbrowser/design/#fingerprinting-linkability, 2014. Accessed on 2014-11-30.

[26] A. Narayanan and V. Shmatikov. How To Break Anonymity of the Netflix Prize Dataset. *CoRR*, abs/cs/0610105, 2006.

[27] N. Nikiforakis, W. Joosen, and B. Livshits. PriVaricator: Deceiving Fingerprinters with Little White Lies. In A. Gangemi, S. Leonardi, and A. Panconesi, editors, *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 820–830. ACM, 2015.

[28] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 541–555, Washington, DC, USA, 2013. IEEE Computer Society.

[29] nugg.ad AG. Predictive Behavioral Targeting. https://www.nugg.ad/en/smart-audience-platform/audience-toolbox.html, 2016. Accessed on 2016-02-19.

[30] P. Ohm. Broken Promises of Privacy: Responding to the Surprising Failure of Anonymization. *UCLA Law Review*, 2009.

[31] L. Olejnik, G. Acar, C. Castelluccia, and C. Díaz. The leaking battery: A privacy analysis of the HTML5 Battery Status API. *IACR Cryptology ePrint Archive*, 2015:616, 2015.

[32] M. Perry. Apply third party cookie patch. https://trac.torproject.org/projects/tor/ticket/3246, 2011. Accessed on 2014-07-10.

[33] A. Phillips and M. Davis. BCP47 - Tags for Identifying Languages. *IETF Trust*, 2009.

[34] P. Stone. Pixel Perfect Timing Attacks with HTML5. White Paper, 2013. http://contextis.co.uk/files/Browser_Timing_Attacks.pdf.

[35] The Chromium Project. Chrome Browser. http://www.chromium.org/Home, 2015. Accessed on 2015-02-10.

[36] The Tor Project. Tor Browser. https:

//trac.torproject.org/projects/tor/ticket/5798#comment:13, 2015. Accessed on 2015-01-11.

[37] H. Tillmann. Browser Fingerprinting Dataset. http://bfp.henning-tillmann.de/?lang=en, 2014. Accessed on 2014-09-10.

[38] C. F. Torres, H. L. Jonker, and S. Mauw. FP-Block: Usable Web Privacy by Controlling Browser Fingerprinting. In G. Pernul, P. Y. A. Ryan, and E. R. Weippl, editors, *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*, volume 9327 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2015.

[39] T. Vissers, N. Nikiforakis, N. Bielova, and W. Joosen. Crying Wolf? On the Price Discrimination of Online Airline Tickets. In *7th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2014)*, Amsterdam, Netherlands, Jul 2014.

[40] M. Wu, R. C. Miller, and S. L. Garfinkel. Do Security Toolbars Actually Prevent Phishing Attacks? In R. E. Grinter, T. Rodden, P. M. Aoki, E. Cutrell, R. Jeffries, and G. M. Olson, editors, *Proceedings of the 2006 Conference on Human Factors in Computing Systems, CHI 2006, Montréal, Québec, Canada, April 22-27, 2006*, pages 601–610. ACM, 2006.

8. APPENDIX

8.1 Flash and System Font Protection Mechanisms

DCB enables a general Flash and system font fingerprinting protection for both $N:1$ and $1:N$ strategies. In the following, we give details on the implementation.

Flash Player Capabilities-Class: This class provides a wide range of fingerprintable system information, such as screen information or Windows version, which are not covered by other anti-fingerprinting tools when Flash is activated in the browser. The Capabilities functionality is compiled and encoded as hex strings inside the *pepperflash.dll* file. To ensure fingerprinting protection we therefore use Python to manipulate the Capabilities-Class on all system parameters that are accessible by Flash. The parameters are changed to match the values that are retrievable by JavaScript (like User Agent, screen object or others), and were already modified by the DCB configuration.

System Fonts: To counter font probing via CSS and JavaScript, we internally change requested font names either to existing fonts or non-existing fonts, depending on the strategy, whether we want to fake the existence or non-existence of a system font. For example, we might hide the existence of the font *Comic Sans* by changing the request into a fictive font '*aaa*'. Internally, the font will not be found and therefore the browser's default fallback font will be used, even the font is actually installed on the system. Analogously, to fake the existence of a non-installed font, DCB automatically renders the text with another existing system font such as *Comic Sans*. Since *Comic Sans* is not the fallback font, the fingerprinter will then assume the existence of the requested font. We also manipulate the list of fonts returned by Flash. Here, we either add new font names or filter out existing names from Chromium's internal font list (see Sections 8.2 and 8.3 for a detailed description).

8.2 Specific N:1 Implementation

In this section we describe the specific method how fingerprinting features are manipulated in the $N:1$ strategy.

Screen Information: In order to achieve a common screen configuration, the server will determine the most frequent screen resolution, color depth, and pixel depth for every configuration group. The server will also choose among those screen information that fit with the available users' screen (monitor).

Browser Language: Browsers using the same main language and operating system are joined into a configuration group. All other languages that are part of the *HTTP_ACCEPT_LANGUAGE* header commonly have a lower priority (q value) than the main language. We adapt a language with a lower priority only to a configuration group in case that at least 3/4 of all browsers in the configuration group share that common language.

Plug-in Information: Since we want to create configuration groups that are as large as possible, our goal for the $N:1$ strategy is to reduce the revealing of information to a minimum, making it easier to join browsers into anonymity groups. For this reason we disable all plug-ins that are not part of every browser of a configuration group, which consequently can result in disabling all plug-ins that are not shipped with Chromium. Along with this step, the most common name and description of a plug-in is adopted in order to equalize the possible version information. Note that we could have considered plug-ins as usability constraint and therefore avoided the chance of disabling any plug-ins. However, as this would reduce the size of the configuration groups we decided against it. Yet, anti-fingerprinting tools with a large user base could choose to handle plug-ins as constraint feature.

User Agent: For every configuration group the most common Chrome version, WebKit/Blink version and Windows architecture are selected and then shared between all browsers of that group.

System Fonts: The list of fonts is set to the common intersection of fonts of all browsers in the configuration group. To share and apply the list of fonts among the browsers of a configuration group, we use the fact that the Windows version will be the same for all browsers sharing the same configuration. Here, the final font configuration only contains the fonts delivered with the initial operating system installation (see Appendix 8.1).

Time and Date: In order to set a common time zone offset for every browser of a configuration group, the most frequent offset of all browsers in group is selected.

8.3 Specific 1:N Implementation

DCB applies the following fingerprint feature modifications in the $1:N$ strategy to achieve the goal of a diverse browser configuration.

Screen Information: The screen resolution, color depth, and pixel depth, are randomly selected among the already observed (prestored) values. Regarding the available resolution, the height of the selected screen resolution is reduced according to the height of the taskbar of the selected Windows version. For example in Windows 7 the taskbar might have a height of 30 or 40 pixels depending on the user choice. If more than one value is possible, the height is chosen randomly.

Browser Language: While Flash and *navigator.language* only return the main language of the user, we do not need to change these values, as we keep the main language for usability reasons. We only manipulate the list of additional languages returned by *HTTP_ACCEPT_LANGUAGE*. Languages are separated by a comma, the language code and priority q are separated by a semicolon. Note that the priority of the main language with the highest priority of 1 is omitted in the language header. Besides the main language and every language with a priority $\geq$0.8, we randomly select up to three additional languages with a random quality between 0.7 and 0.1

(steps of 0.1). We only add language codes according to BCP 47 [33] in order to prevent implausible language codes. An exemplary language acceptance header and the priority of the corresponding languages looks like: de-DE,de;q=0.8,en-US;q=0.6,en;q=0.4

Plug-in Information: If at least 5 different versions of a plug-in are available in the database, the server will select a plug-in description randomly. Otherwise, the server searches for version numbers in the name and description and manipulates the minor version numbers. If the minor version is not available, we randomly change the provided version number.

User Agent: We do not hide the actual browser vendor, as it could be detected using vendor specific features [28]. Instead, we manipulate the version numbers according to real browser which provides the necessary fingerprint diversity. The WebKit/Blink and Chrome version numbers are changed according to the algorithm used to manipulate the plug-in version numbers (described above). The Windows version is randomly chosen among the already observed and pre-stored Windows versions. The Windows architecture is chosen from the fixed list of possible, vendor specific values, which depend on the processor and the bit version of the operating system.

System Fonts: The list of fonts contains 90 to 320 randomly chosen fonts from the list of previously observed fonts (i.e., from the configuration server), along with the fonts that are shipped with the specific Windows version. Therefore, only realistic font names will be observed by a fingerprinter.

Time and Date: The function *getTimezoneOffset()* is commonly used as a fingerprinting feature. In order to maintain consistency, we change the time zone offset along with all other time and date information retrievable by the *Date* class.

8.4 Weaknesses in Counter Detection Strategies of Canvas Manipulation

Although the below described strategies could be applied to hide the manipulation of *toDataURL()* or *getImageData()*, the weaknesses that may lead to their detection still exist:

(1) One may store hashes of previously generated canvases along with their modified version for the duration of a browser session. As soon as an image with the same hash is requested, the previous modification of the image is returned to prevent the detection of differences. Yet, this approach would be detectable if a fingerprinter would add a localized change to the canvas and only compare non-effected parts to a previously returned output. Since this approach calculates the hash of the overall image, a new modification is applied causing the hash to change, unmasking the anti-fingerprinting measures.

(2) To circumvent the comparison of localized changes, an anti-fingerprinting algorithm could modify the localized changes and copy the old modification of those parts that are exactly the same. Still, the problem exists if many different changes are added and the fingerprinter would compare a partial hash of those areas.

(3) In order to avoid the detectability of small modifications, an algorithm could store all distinct canvases of a session. When a new canvas is about to be manipulated, all areas of similarity of prior canvases need to be re-placed with their respective recorded images. Newly observed areas would be then modified separately with random noise. Again, this approach could be detectable when a fingerprinting script would render the same image information twice on one image. If the image was new to the algorithm, it would randomize both parts differently.