Analysis and Implementation of Browser Fingerprinting Techniques and Defenses

Kevin Valakuzhy
kevinv@gatech.edu
Georgia Institute of Technology

Mohammad Zaid Khaishagi
mkhaishagi3@gatech.edu
Georgia Institute of Technology

Yoon Jae Lee
yoon.lee@gatech.edu
Georgia Institute of Technology

Shea Wells
wells.shea@gatech.edu
Georgia Institute of Technology

Guoqiang Zhang
gzhang315@gatech.edu
Georgia Institute of Technology

***Abstract*—Browser fingerprinting is an advanced device-identification technique employed for tracking users across the internet. While its use has many benefits to website owners, such as providing site-visitor demographic and interaction data for more informed decision making, browser fingerprinting poses a threat to internet-user privacy. In this paper we explore browser fingerprinting techniques and countermeasures implemented by modern browsers to address them, implement a browser extension to reduce user identifiability, and then analyze the extension's behavior on real websites.**

I. Introduction

Tracking on the internet has a long history of privacy concerns that often go under the radar, so it is often not obvious to the everyday internet user how to avoid it. Tracking typically involves collecting information about user identity and interaction on a website or across multiple websites.

Collecting this information is important for many websites. It provides a way for site owners to monitor site visitor information and activity, allowing site owners to make more informed decisions concerning the site itself as well as the underlying business. Tracking services like Google Analytics provide ways to see how people find and use a site, along with who and where these people might be [1]. This is often helpful for making improvements to site functionality, design, or even content, such as through targeted ads or personalized experiences.

Early tracking techniques often used cookies, which are text files created by a website and stored in the browser to maintain information regarding identity. However, this data can be removed by the user to prevent their use in tracking. Consequently, more advanced cookie techniques allowed storing cookies in non-traditional places, making removal more complicated. Once these techniques became widely known, efforts were made to prevent the usage of cookies by third party tracking companies, with some browsers suggesting to remove them altogether.

With the declining efficacy of cookies came browser fingerprinting, a robust user-identification method that works even when cookies cannot be read or stored. A browser fingerprint is made up of data collected by a website from the user's browser. This data is normally used to ensure proper site functionality, however, when grouped together, this data can be used for tracking. Commonly used data includes information about graphics, audio, fonts, plugins, memory, browser version, and more. Once a website has this data, it is linked together to create a unique identifier for the visitor. Once a fingerprint is gathered, it can be used to track user behavior throughout a single site, or even across different sites.

Browser fingerprinting raises privacy concerns since many users could be identified or tracked across sites, with their browsing behavior potentially stored and linked back to them. Even if users take simple steps to increase privacy, like using private browsing or clearing cookies and caches, browser fingerprinting can continue to track them.

We decided to look deeply into the inner workings of browser fingerprinting and understand not only fingerprinting techniques, but also how to counter them. We began our project with learning how to implement browser fingerprinting techniques by building a website that utilizes a few common ones. Then we looked into the techniques modern browsers use to prevent browser fingerprinting and compared their effectiveness in doing so. Next, we implemented anti-fingerprinting techniques ourselves in a Google Chrome extension with the

goal of reducing user identifiability. Finally, we analyzed the effectiveness of our extension, as well as the side effects it has on browsing the web. The code for our extension can be found in our github repository [2].

II. Related Work and Motivation

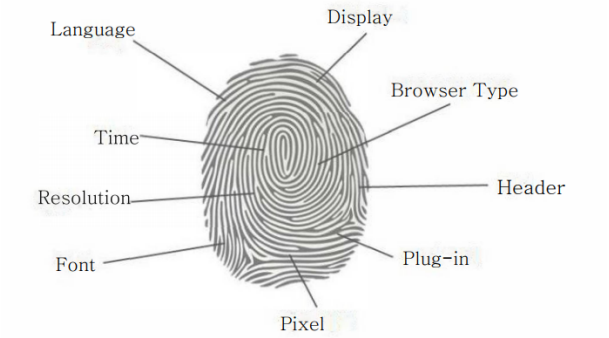

Fig. 1. Information from many data sources is collected to generate a user's digital fingerprint

Historically, device fingerprinting was adopted by banking websites to prevent fraud. It was used to track user behavior to determine whether activities came from trusted devices or fraudulent actors. The technique does not require a tracker to set any state in the user browser, but trackers identify the user by combining device properties. Studies from Englehardt and Narayanan [3] show that within a sample of 100,000 browsers, around 80% of both desktop and mobile device fingerprint are unique.

Studies from Englehardt and Narayanan also show a continual increase in third-party tracking and HTTP tracking techniques across the web. The study indicates that Google can track users across nearly 80% of sites through third-party domains and HTTP requests [3]. Even though these studies increased awareness of the techniques, they are still being used for tracking. To address these privacy issues, browsers have developed techniques to block the collection of fingerprinting information for use in identifying users.

III. Approach

In this section, we outline the approach we took to better understand the features that can be used to perform browser fingerprinting, how to counteract them, and how to measure the impact of our approach.

A. Browser Fingerprinting Techniques

In order to obtain a better understanding of fingerprinting techniques, we developed a website

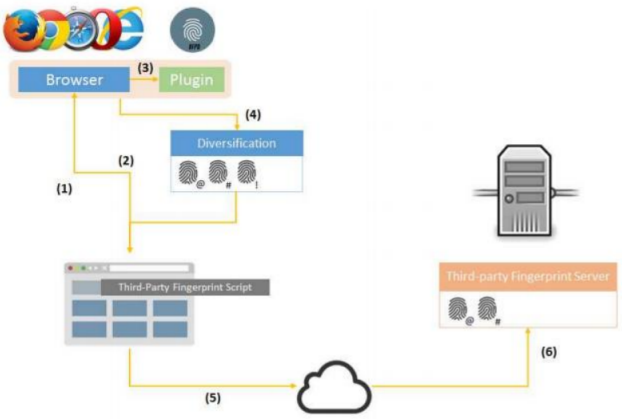

Fig. 2. Browser Finger Printing Diversification Flow

that captures various features from a visiting user's browser.

Before implementing the techniques, we needed to identify the different features that websites can use for fingerprinting. We enumerated these features using the following testing websites which perform browser fingerprinting:

- CoverYourTracks (coveryourtracks.eff.org)
- BrowserLeaks (browserleaks.com)
- FingerprintJS (fingerprintjs.com)

There was some overlap between the fingerprinting features shown by these services. For FingerprintJS, it does not show the specific features it captures but only displays an identifier that is associated with a visiting user. Since FingerprintJS has a free open source version, its code provides helpful insights into the approaches and the methods that can be used for fingerprinting. However, the service we relied on the most was CoverYourTracks, which provides an analysis of the level of protection a user has against fingerprinting and examples of data used to generate the user's fingerprint.

After reviewing the different approaches to fingerprinting, we attempted to replicate these on our website. To maintain a reasonable scope for our project, we aimed to replicate only features displayed by CoverYourTracks. The status of our implementation for each of these is also shown in Figure 3.

Implementing these gave a concrete view of the techniques used in practice in order to fingerprint users. This knowledge was then used to build a tool that provides protection against fingerprinting as described in later sections.

Coveryourtracks	Our website
User Agent	Yes
HTTP_Accept Headers	No
Browser Plugin Details	Yes
Time Zone Offset	No
Time Zone	Yes
Screen Size and Color Depth	Yes
System Fonts	Yes
Cookies Enabled	Yes
Limited Supercookie Test	No
Hash of Canvas Fingerprint	No
Hash of WebGL Fingerprint	No
WebGL Vendor and Renderer	Yes
Do-not-track Header Enabled	Yes
Language	Yes
Platform	Yes
Touch Support	Yes
Ad Blocker Used	No
Audiocontext Fingerprint	No
CPU Class	No
Hardware Concurrency	Yes
Device Memory	Yes
Network Details (not on Coveryourtracks)	Yes

Fig. 3. CoverYourTracks fingerprint details

B. Browser fingerprinting defenses

After investigating fingerprinting techniques, we looked at some popular browsers and their countermeasures against these approaches. Specifically, we looked at the data collected by CoverYourTracks and the documentation and source code, as available, for Chrome, Firefox, Brave, and Tor Browser.

From our analysis, we can summarize three main approaches that browsers adopt to counter fingerprinting. These are given as follows:

Anonymize. This approach manipulates browser features so that they assume the most common value across all users. For example, the most common operating system for PCs is Windows, so a browser using anonymization would modify all user agent strings to indicate Windows regardless of whether the user's operating system is Linux, Windows, MacOS, etc. This is the strategy used by the Tor browser.

The main benefit of this approach is websites that attempt to perform fingerprinting will not have enough information to uniquely identify any given user. However, to be most effective, this approach requires knowledge of the statistics of common user characteristics, which needs to be collected from the user base. In addition, if only selected values are modified and the overall characteristics of the browser are not considered, anonymization may make the user more unique. For example, if a browser claims to be running on Windows, but indicates that it has fonts that are only installed on a MacOS version of the browser, the combination of features may be uncommon and, therefore, useful for tracking.

Randomize. In this approach, the browser features are set to random, reasonably likely values that may or may not match the user's actual features. For example, the device memory that the browser is able to detect may return a random value. Brave takes this approach with certain features, such as randomizing hardware concurrency.

One benefit of this strategy is that this does not require statistical data about user features and can be used to prevent a website from tracking a user across multiple visits. Even though a set of random values may uniquely identify a user during their initial visit to a website, the website will not be able to track the user in subsequent visits as the features used for fingerprinting would have changed. However, randomization must be done carefully since just the presence of unreasonable values could distinguish a user, and randomized values can negatively impact usability. For example, if the browser uses randomization to indicate it is running on a mobile platform while it is actually running on a desktop, the layout of the webpage might break.

Disable. A browser may also choose to simply block any API calls that are made for information that can be used for fingerprinting. This would make fingerprinting difficult by preventing any leakage of user characteristics to the website. Firefox does this for some features such as browser plugins.

Although this approach is effective at blocking fingerprinting vectors, it is limited in its applicability. Some information is necessary in order to optimize interactions with a website on different devices with different capabilities. For example, by omitting the user agent string in an HTTP request we found that some websites defaulted to sending content for an android browser, even when we were browsing on a desktop.

In the browser extension we implemented to defend against browser fingerprinting, we considered these three approaches. For the purpose of this project/paper, we decided to implement the randomization approach. The specific implementation details and results are described in the following section.

C. Anti-fingerprinting browser extension

In this section we will discuss how we develop and use a Chrome extension to randomize the browser fingerprint.

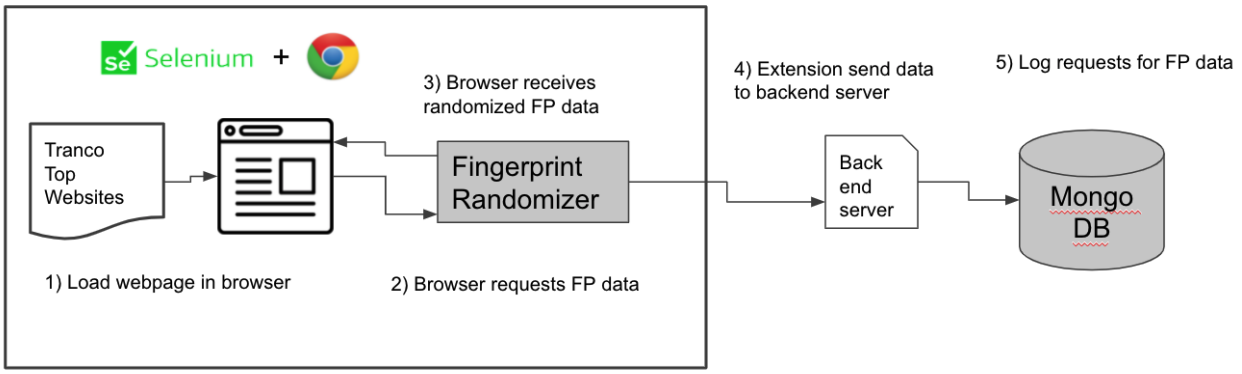

Fig. 4. Flow of Data Collection

JavaScript Property Overwriting. Most of the metrics used for fingerprinting are collected via calling the JavaScript in the browser. For example, the JavaScript shown below can get the browser's screen size.

```
var width = window.screen.width;
var height = window.screen.height;
```

The idea here is to directly set **window.screen** to a random value before the website calls it. JavaScript provides a function **Object.defineProperty** to achieve this. The following code shows how to overwrite **window.screen** to a value we want.

```
Object.defineProperty(
    window,
    "screen",
    {
        value: {
            width: randInt,
            height: randInt,
        }
    }
)
```

Now the problem is how can we overwrite the functions before the website calls them. Luckily, the Chrome extension provides a "run_at" [4] option to inject the script before the document starts loading.

Canvas Fingerprinting. Canvas fingerprinting is usually performed by creating an invisible canvas element and draw a complex shape or text using JavaScript on it. A hash value can be generated based on the pixels information. Usually due to the difference of the hardware, the same shape will result in different hash values on different devices [5]. One way to randomize the hash value is to add random pixels when drawing the shape. The good side of this approach is introducing minimum changes to existing hash function. But the drawback is that the random pixels are visible to the users. Another way is to directly overwrite the hash function. We are using the second way.

The function usually used to get the canvas data is **toDataURL**, called from **HTMLCanvasElement.prototype**. As it can be seen, this is a prototype function of type **HTMLCanvasElement**. A prototype function is a function which all the objects of **HTMLCanvasElement** type will inherit. By overwriting the **toDataURL** function, the change will be applied to all the canvas objects.

Alerting the User. Another feature of our extension is to alert users when any of the fingerprinting functions or properties we randomized are called by the website. Example code is shown below with the screen size as an example.

```
Object.defineProperty(
    window,
    "screen",
    {
        value: {
            get width() {
                alert("screen.width");
                return randInt;
            },
            get height() {
                alert("screen.height")
                return randInt;
            }
        }
    }
)
```

For functions this is easy to achieve, as we can add the alert script in the function body. It is a slightly more complicated for properties because these are just values. Here we make use of the getter [6] pattern which allows accessing the property in the normal way, but under the hood we can define it as a function and add the alert script in the function.

Data collection. Another important feature of our extension is to collect data for analysis. It consists of three important parts. An Ajax call in the Chrome extension, a back end server for receiving data from the extension, and MongoDB for data storage.

D. Ajax Call in Chrome Extension

Ajax calls are used to send data to the back end server. Ajax is short for Asynchronous JavaScript and XML. With Ajax, we can send data from client to a server asynchronously without blocking other scripts from running [7]. By adding the following code snippet in the functions we have overwritten, the data will be automatically sent to **http://127.0.0.1:3000/analysis**.

```
fetch(
    'http://127.0.0.1:3000/analysis',
    {
        method: "POST",
        headers: {
            'Content-Type':
            'application/json'
        },
        body: JSON.stringify({
            "site":
                window.location.origin,
            "fnCall": fnName
        })
})
.then(data => console.log(data));
```

E. Back End Server and CORS

CORS is short for Cross-Origin Resource Sharing. It is an HTTP-header based mechanism that allows a server to indicate any other origins (domain, scheme, or port) than its own from which a browser should permit loading of resources [8]. Because we are collecting statistics from all websites, the back end server must set the CORS headers to allow any origin. So the browser will not block the Ajax request. After receiving the data from client, the back end server saves the data into MongoDB for further analysis.

F. Automatic Data Collection

With all preparation, we are ready to collect some statistics. We use a python-selenium Chrome driver to help us automatically load websites. Selenium is an automation tool used for automate web testing. It provides a rich API calls to simulate user behavior. For our use case, we only need to use selenium to automatically open our web page and load the extension we created. The extension will send the statistics to the back end server if the functions we are tracking are called. Figure 4 summarizes our data collection flow.

IV. Findings

A. Website

The techniques we identified and used for capturing each of the features we implemented is explained next along with their effectiveness.

User Agent: This can be captured by accessing the correct attribute in the ‘navigator‘ object. It is accessible simply by calling ‘navigator.userAgent‘.

Browser Plugin Details: This can also be accessed using the navigator object, using the call ‘navigator.plugins‘. This approach is easy as well as effective in capturing the plugin details of the browser. However, some browsers offer protection against this.

Time Zone: This can be accessed using the ‘Intl‘ object. Accessing the functionality available in ‘Intl.DateTimeFormat()‘ gives this value reliably.

Screen Size and Color Depth: The ‘screen‘ object provides APIs to capture both the screen dimensions as well as the color depth. For example, ‘screen.colorDepth‘ gives the color depth.

System Fonts: The system fonts need to be checked individually. So, a span element is created using JavaScript and populated with some default text. Then, the font for this element is changed iteratively selecting from a predefined array of possible system fonts. Upon changing the font, if the dimensions of the span element change from the default values, then it can be inferred that this specific font is available on the system. The list of fonts used in our implementation is taken from the Windows 10 and MacOS font lists [9] [10]. We were able to capture almost all of the fonts that were captured by Coveryourtracks.

Cookies Enabled: This can again be accessed through the ‘navigator‘ JavaScript object by calling ‘navigator.cookieEnabled‘.

WebGL Vendor and Renderer: This information is gathered by first creating a canvas element in the document using JavaScript and then calling the getContext() method for it with the

argument 'webgl'. This can then be used to fetch the details using the getParameter() and the getExtension() methods.
Do-not-track Header Enabled: The 'navigator' object provides access using 'navigator.doNotTrack'.
Language: It can be accessed using 'navigator.languages'.
Platform: This can be determined by analysing the string returned by 'navigator.appVersion'.
Although the platform is not returned directly, it can still be easily determined by parsing the string.
Touch Support: This is checked by testing the values of multiple variables. The relevant variables indicating touch support are in the window, navigator and the window.navigator objects.
Hardware Concurrency: The 'navigator' object provides access though the 'navigator.hardwareConcurrency' property.
Device Memory: This can also be accessed through the 'navigator.deviceMemory' property.
Network Details (not on Coveryourtracks): This can be detected by using the 'navigator.connection' object.

It is noteworthy that most of these were fairly straightforward to capture using simple JavaScript.

B. Browser defenses

Table I indicates defenses used by browsers to prevent fingerprinting. We notice that browsers have wildly different coverage in the information they protect against fingerprinting. Moreover, the default settings of Chrome, the most widespread browser, reveals all the parameters we checked, including Fonts, Language, WebGL, and Audio information. On the other hand, the Brave and Tor browsers provide defenses across the board, suggesting that their fingerprinting avoidance should be much more effective. The table shows that one's choice in browser can have serious privacy consequences.

C. Anti-Fingerprinting Extension

We were able to run our chrome extension on top websites, as identified by Tranco [11], to collect information on the usability of our extension and popularity of APIs that have fingerprinting potential. Furthermore, we used our extension on the CoverYourTracks website, which indicated that we were successfully able to randomize our fingerprint, a result that Chrome normally does not achieve, but is seen in a more privacy focused browser like Brave.

D. Usability Analysis

Using our database to store results from our selenium setup, we identified the difference in the number of errors when we browsed the internet with and without our extension. We found that over half the time we didn't change the number of errors that were produced. About 9% of websites we decreased the number of errors, but we believe this could be due to expected variability when accessing a webpage, or a new error preventing more from being produced later on.

While it is possible to record the type of error as well, we found it difficult to understand how our extension caused errors. This is mostly because the errors occur in a different location from where the randomized data was introduced.

E. Case Study: Video Conferencing Errors

With our usability analysis, although we got an idea of how many websites were impacted, we were not sure what the impact of the errors were. To collect more information, we turned on the extension during our normal day-to-day usage. For most websites, even if errors did occur, we did not perceive a decrease in functionality. However, we did notice for websites that used video, such as Bluejeans and Google Meet, that enabling this extension would prevent video conferencing from working. We tried to investigate the causes for these issues, but it was clear that the errors were not directly connectable to the information that was randomized.

F. Case Study: AudioBuffer API

We also looked at the APIs used for fingerprinting to see how common they were across the top websites. One API that was relatively rare was AudioBuffer-getChannelData. This allows inspection of audio information before the audio is played out loud. It is clearly a relatively rare API, but is also key to the process of audio fingerprinting, which requires analyzing minute differences between how audio is generated on different machines. Because this API has a non-tracking use case, as all APIs do, it is difficult to identify whether it is being used for fingerprinting in a certain context. Many of the websites that we found to utilize this API contain videos, meaning it could have been used for its originally intended purpose.

V. Discussion

Perhaps our most concerning finding was how easy fingerprinting techniques are to implement. We

TABLE I
COMPARISON OF BROWSER FINGERPRINTING DEFENSES

Data Type	Chrome	Firefox	Brave	Tor
Fonts	Reveal	Only Default Font (Disable)	Default Font (Disable)	Default Font (Disable)
Language	Reveal	Reveal	Reveal	Reveal
WebGL	Reveal	Reveal	Anonymous	Anonymous
Audio	Reveal	Random	Random	Permission
Hardware Info	Reveal	Reveal	Random	Anonymous
Browser Plugin	Reveal	Anonymous	Random	Anonymous

TABLE II
NUMBER OF NEW ERRORS CAUSED WHEN BROWSING TOP WEBSITES WITH OUR EXTENSION ENABLED

# JS Errors Introduced	# Websites (N=228)
Decreased Errors	20 (9%)
No New Errors	128 (56%)
Between 1 and 10	34 (15%)
Between 11 and 100	20 (9%)
Greater than 100	26 (11%)

TABLE III
TOP WEBSITES THAT USE THE AUDIOBUFFER-GETCHANNELDATA API, AS WELL AS THE WEBSITE CATEGORY.

Website	Categories
www.yy.com	Streaming Media & Downloads
www.bloomberg.com	News
www.sciencedirect.com	Education
www.pornhub.com	Pornography/Sexually Explicit
www.bilibili.com	Streaming Media & Downloads
www.taboola.com	Business
www.zhihu.com	Forums & Newsgroups
www.binance.com	Computers & Technology
www.prnewswire.com	News

were easily able to find and incorporate JavaScript into our website that could be used to generate fingerprints. Libraries such as fingerprintJS also offer easy to use APIs that simplify the process of marinating records of fingerprints for users. It's unclear to us how widespread fingerprinting is, but it is obvious that there is very little preventing most websites from collecting this information, especially if they were previously using cookie IDs to track users.

We quickly learned there are real usability downsides to using privacy prevention tools, but it is difficult to quantify how much of an impact these usability downsides are.

Moreover, even if the current fingerprinting techniques are defeated, it is unclear whether those will work for new ones. Recently, a technique involving the use of favicons was discovered [12], allowing for user identification through the seldom cleared favicon cache. This should be a reminder that protected privacy is an ongoing cat-and-mouse game that results in continuous innovation on both sides.

Innovations in advertising technology may be an alternate source of hope for consumer privacy and possibly privacy in general. Brave promoted it's own ad system which limited information that advertisers could collect from users by relying on client side machine learning [13]. Google Chrome is pushing forward a privacy sandbox that aims to keep more user information on device, utilizing a Privacy sandbox [14] to prevent leakage of user information. In addition, Chrome is also pushing forward on Federated Learning of Cohorts [15], which allows advertisers to target groups of similar people while keeping user data on the user's machine.

VI. FUTURE WORK

To improve upon our extension for popular use, we would need to gather more information on why the errors we measured were caused. However, it may be difficult to identify the problem from just the error, as where the code crashes and where the error message is printed may be far apart. Instead, we could browse the web multiple times, turning on each one of our randomization options to see how many errors are caused for each of the individual APIs that we randomize.

Furthermore, we could improve our extension's user experience if we could tell the user, with some degree of confidence, whether some information that is being requested to be used for fingerprinting. Not only could that allow the extension to reduce false positives, we could give the user the choice of allowing information when it is requested, similar to how permissions on mobile devices work now. We believe that a model based on the rarity of an API along with the context of the API, such as where the JavaScript file was sourced from, may provide signals for usage of APIs for fingerprinting.

VII. CONCLUSION

In this work, we investigated browser fingerprinting techniques that are currently being

used in the wild, and built our own website to capture these features while demonstrating that these techniques were simple enough for them to be widely adopted. Then we investigated techniques browsers use to protect their users from being fingerprinted. Using these findings, we built a fingerprint randomization extension for Google Chrome which successfully randomized our fingerprint, preventing test websites such as CoverYourTracks from establishing a consistent fingerprint for our device. Finally we discussed where we believe our extension could be improved, and how we see privacy and advertising technology evolving as we move forward.

References

[1] B. Clifton, *Advanced Web Metrics with Google Analytics*. John Wiley Sons, 2012.

[2] G. Zhang, *Gzhang315/cs6262*, https://github.gatech.edu/gzhang315/cs6262.

[3] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2016. DOI: 10.1145/2976749.2978313. [Online]. Available: https://www.cs.princeton.edu/~arvindn/publications/OpenWPM_1_million_site_tracking_measurement.pdf.

[4] Google, *Content scripts*. [Online]. Available: https://developer.chrome.com/docs/extensions/mv2/content_scripts/.

[5] Wikipedia, *Canvas fingerprinting*. [Online]. Available: https://en.wikipedia.org/wiki/Canvas_fingerprinting.

[6] M. W. Docs, *Getter*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get.

[7] Wikipedia, *Ajax (programming)*. [Online]. Available: https://en.wikipedia.org/wiki/Ajax_(programming).

[8] M. W. Docs, *Cross-origin resource sharing (cors)*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

[9] Apple, *System fonts*. [Online]. Available: https://developer.apple.com/fonts/system-fonts/.

[10] Microsoft, *Windows 10 font list*. [Online]. Available: https://docs.microsoft.com/en-us/typography/fonts/windows_10_font_list.

[11] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A Research-Oriented top sites ranking hardened against manipulation," Jun. 2018. arXiv: 1806.01156 [cs.CR].

[12] K. Solomos, J. Kristoff, C. Kanich, and J. Polakis, "Tales of favicons and caches: Persistent tracking in modern browsers," in *Proceedings 2021 Network and Distributed System Security Symposium*, 2021.

[13] Brave, *An introduction to brave's In-Browser ads*, https://brave.com/intro-to-brave-ads/, 2020.

[14] J. Schuh, *Building a more private web*, https://www.blog.google/products/chrome/building-a-more-private-web/, 2019.

[15] *Evaluation of cohort algorithms for the FLoC*, 2020.