A Comparative Study on Analyses of Browser Fingerprinting

by

Timothy Stephenson

A thesis submitted to the faculty of Wesleyan University in partial fulfillment of the requirements for the Degree of Bachelor of Arts with Departmental Honors in Mathematics and Computer Science

Middletown, Connecticut April 2023
This thesis is dedicated to my swim coaches I have had throughout the years. I would be a fraction of the individual I am today, if not for the mentorship and character building that has been provided to me by Mario Francisco Sobrinho, Peter Solomon, and Scott Ferrigno. Being a competitive swim coach can often be a thankless job, especially in contrast to the impact they have on the likes of myself and countless others.
Acknowledgements

Above all, I would like to thank my advisor Norman Danner, who has been instrumental in his guidance, foresight, knowledge. Without Professor Danner I would have never finished the research I have accomplished. He has provided unwavering support and structure to myself, who would been lost without his help in prioritization. I am grateful for my reading committee, Norman Danner, Sebastian Zimmeck, and Daniel R. Licata. I am appreciative to all of my fellow Computer Science majors, especially those who work in Exley 341 with me. I am beyond thankful for the privilege of unwavering support from my parents, Gregory and Rebecca. In my time at University I have seen them the least, but I have felt them the most. Lastly, I am thankful for my siblings, David and Hannah who are my swimming partners, my running partners, and my coding partners.
Abstract

Browser fingerprinting has emerged as a stateless tracker of web users. It utilizes the entropy of a device’s hardware and software configuration to associate a web user with their (potentially unique) fingerprint and track them throughout the web. Significant prior research has worked towards the task of the analysis/detection of browser fingerprinting. We explore the effectiveness of dynamic and lexical analysis methodologies to classify the occurrence of browser fingerprinting on a 10,000 site web crawl. We showcase the current state of the effectiveness of these methods, and implement an improvement to a lexical analysis by unescaping string literals based on our observations.
## Contents

Chapter 1. Introduction 1
   1. Browser Fingerprinting 1
   2. Browser Fingerprinting Analysis 3
   3. Contribution 6
   4. Motivation for comparison of browser fingerprinting analyses 8

Chapter 2. Background on Browser Fingerprinting 9
   1. History of Browser Fingerprinting 9
   2. The Uniqueness of a User’s Browser Fingerprint 10
   3. Browser Fingerprinting Types / Methods of Browser Fingerprinting 12
   4. Applications of Browser Fingerprinting 19
   5. Prior work on analysis/detection of Browser Fingerprinting 21
   6. Mitigation of Browser Fingerprinting 26

Chapter 3. Real World Website Data Gathering 28
   1. Overview 28
   2. OpenWPM 28
   3. Parameters used in OpenWPM 29
   4. Limitations in our data gathering using OpenWPM 30

Chapter 4. Analysis Method and Results 33
   1. Overview 33
   2. Canvas Fingerprinting 34
   3. Canvas Font Fingerprinting 47
   4. WebGL Parameter Fingerprinting 51
CHAPTER 1

Introduction

1. Browser Fingerprinting

Browser Fingerprinting is the extraction of a user’s Browser Fingerprint by a website on the World Wide Web. A user’s browser fingerprint consists of pieces of information the user’s web browser exposes to the web server which differ between web users. Such information consists of user settings, the device’s implementations of low-level functionality, device properties, peripheral device properties, etc. This information is extracted from the client through the JavaScript the web server sends the client, which the web browser then compiles and executes. Additionally, information is extracted from the request headers the client sends the web server. A browser fingerprint helps (possibly different) web servers identify visits by the same user. A browser fingerprint is called a fingerprint because the aggregate of information it is composed of may suffice in identifying a web user. As human fingerprints are the clues a person leaves behind identifying their presence at a location, browser fingerprints are the clues a web user leaves behind identifying their presence on a website.

Performing Browser fingerprinting relies on the web server sending the web client instructions to extract the browser fingerprint from its device, and then send the browser fingerprint back to the web server via a network request. Currently these instructions are in the form of JavaScript, a high-level programming language. This process is an example of client side scripting, a process where the server sends the client a script for the client’s machine to then execute. Client side scripting in the context of the World Wide Web can perform a plethora of functionality, such as modifying the web page, reacting to user inputs and/or behavior, performing network requests with any web server, executing additional client side scripts, etc. Currently, nearly all of the client...
side scripting on the World Wide Web is done via JavaScript. JavaScript’s core API semantics and language syntax are determined by the EMCAScript standard.[41] The core API semantics include built in objects and types, operators, etc. The core API semantics do not include any I/O.[31] JavaScript’s interaction with I/O and the browser window are determined by JavaScript Web APIs, or the Web API. The Web API is a set of functions and objects made available by the web browser to JavaScript being executed. These functions and objects include the Document Object Model (DOM), access to properties about the client’s machine, an interface with local storage, an interface for network requests, etc. The DOM is an tree-like object representation of the document visible to the user. The DOM functions as a interface for JavaScript scripts to interact with the page. This Web API supplies the information that constitutes the browser fingerprint. Standardization of the Web API across web browsers is set by the World Wide Web Consortium (W3C). Web browsers independently decide their conformance to W3C standards, including the Web API, weighing the pros and cons of a feature.

Certain pieces of the Web API provides information which differs from user to user. Examples of this include:

window.matchMedia("(dynamic-range:high)") which returns whether the user is using a HDR (high dynamic range) display and window.navigator.hardwareConcurrency which returns the number of logical processors available on the user’s machine. Properties apart of the Web API exist because they provide legitimate functionality towards creating a website. Being aware if a user’s display can output HDR content allows a site to conditionally provide the user with HDR content. Knowing the number of logical processors a client’s device has aids in efficiently parallelizing a compute heavy task. However, these distinctions revealed by the Web API assist in identifying users, and as a result, contribute to the browser fingerprint. This is what we call information querying. It is a simple form of browser fingerprinting that utilizes the information the Web API transparently makes available. Certain pieces of the Web API utilize lower level system functionally that slightly differ in functionality based of the system’s implementation of such functionality. Examples of this include the Canvas HTML element,
1. INTRODUCTION

and associated Canvas API. The Canvas API allows a script to draw shapes and letters onto a Canvas element, and then query the resulting image data. The resulting image data differs between different user systems due to differences in the translation of the high-level instructions to low-level instructions for the rendering device (e.g. GPU) to perform primitive graphical operation (e.g. drawing a polygon). These differences in the image data area make up a piece of the browser fingerprint. Exploiting these hidden functionality differences is what we call **exploiting functionality quirks**.

2. Browser Fingerprinting Analysis

Our research implements different analyses of browser fingerprinting, then uses these analyses to classify real world websites, comparing and contrasting the results different analyses returned. A **browser fingerprinting analysis** attempts to conclude on the occurrence of browser fingerprinting on a given website. An analysis of browser fingerprinting may alternatively conclude more specifically on the types of browser fingerprinting occurring. Examples of types of browser fingerprinting include a functionality quirk being exploited (e.g. Canvas), a single information query (e.g. `window.navigator.hardwareConcurrency`), or a set of Information queries (e.g. all of the `window.navigator` properties). Chapter 2.3 goes into detail on the types of browser fingerprinting.

A browser fingerprinting analysis is a function from a input domain to a output codomain. The input domain are the elements an analysis is analyzing over. Examples of domains are websites, specific webpages, specific scripts. The output (codomain) of a analysis is the results the analysis produces. Examples of codomains are a boolean yes/no for fingerprinting, a list of boolean yes/no for different types of fingerprinting or a probability of fingerprinting.

A browser fingerprinting analysis requires data to conclude on the nature of a website. The JavaScript source code that is sent to the client comprises the **static data**. The record of actions performed at run-time comprise the **dynamic data**. An analysis which reasons about the behavior of source code without explicitly executing the source
code is a **static analysis** [29]. A static analysis which reasons about the behavior of the source code from the frequency/occurrences of keywords is a **lexical analysis**. An analysis which reasons about the behavior of source code based on a record of actions taken during the execution of the program, is a **dynamic analysis**.

Both the static and dynamic data can be obtained by instrumenting a user’s web browser (via a browser extension) to collect information as instructed while the user (human or not) browses the web. Such a modification/extension to a web browser may also be combined with a web crawler/driver, to obtain the dynamic and static data from many websites/webpages in an automated fashion. A web crawler/driver is any computer program (rather than human user) which browses the web. The static data is obtained by saving the scripts which get sent from a web server to the client.\(^1\) The dynamic data is obtained by recording specific actions taken by the JavaScript being executed by the web browser. The dynamic data requires making a decision as to what runtime actions to record. Recording every runtime action (variable assignments, function calls, etc) leads to infeasible storage requirements for modern websites. An example of the specific actions to record are all the interactions with a subset of the Web API deemed useful for performing browser fingerprinting. This decision making required at the data collecting step intertwines a dynamic analysis with a specific configuration of collecting dynamic data.

### 2.1. Difficulties of a dynamic analysis

The dynamic data does not include functionality of a website which does not execute at runtime. Thus a key flaw in a dynamic analysis is it can only conclude on the functionality which occurrences during the one execution of the script instrumented. Theoretically, a static analysis may be able to conclude on the functionality of all possible code paths.

Instrumenting the runtime of a script requires a trade off between the breadth of runtime behavior being instrumented, and the storage and speed requirements on the

---

\(^1\)It becomes more complex when instrumenting inline JavaScript as well. We discuss this in further detail in chapter 3.
machine performing the instrumentation. The size of the dynamic data produced by instrumenting the runtime of a script may be magnitudes larger than the script itself, based on the configuration of the instrumentation.

Dynamic data is characteristically more difficult acquire than static data, and more rigid in the information it provides. In terms of difficulty, the static data is trivial to acquire, whereas the dynamic data requires the engineering challenge of setting up a instrumentation of the JavaScript source code. The configuration of instrumentation requires choices at the time of data collection regarding what a analysis will look for in the dynamic data. This forces those constructing a dynamic analysis of repeating the potentially costly step of configuring the dynamic instruction and running a web crawl whenever they change their mind on the desired input data. Beyond the choices as to what Web APIs to instrument, there is more complex dynamic instrumentation that may be desired, such as information-flow data used by [35]. The dynamic data will always represent a loss of information from the static data, as the static data can be used to generate the dynamic data, yet the reverse is not true.

2.2. Difficulties of a static or lexical analysis. In general, the process of determining a possible behavior of a program, using its source code without explicitly running it, is significantly more challenging (and undecidable\(^2\)) than instrumenting a the occurrence of a behavior of interest. Due to it’s undecidability, any static analysis is making an approximation regarding the behavior of a program, almost always reporting claims that are weaker than the truth. However, in comparison, a dynamic analysis is only concluding on the programs behavior for the singular code path taken at run-time based on the inputs/environment. A static analysis may conclude on an approximation regarding all possible code paths.

Beyond the challenges static analyses face in any programming language, the dynamic nature of JavaScript presents an additional challenge in statically analyzing it.

\(^2\)A undecidable problem is one in which it is impossible to construct an algorithm which will always arrive at a correct answer in finite time.
Two JavaScript features which contribute to its dynamic nature are dynamic code execution through `eval()` and `Function()` and computed methods names. `eval()` and `Function()` both take a string as input, parse the string input as JavaScript, and then execute the JavaScript. As the problem of determining the run-time value of a string variable is non-trivial, this adds to the difficulty of determining the functionality of JavaScript source code via static analysis. Computed methods names is the process of calling object methods based on a string input of the method name, rather than a static identifier in the source code. A demonstration of computed method names is given in Figure 1. These dynamic features allow severe obfuscation of the JavaScript source code sent to web users. Code **obfuscation** is the process of modifying the source code to a functionality identical version, but deliberately making the functionality opaque to a (typically human) observer. Code **minification**\(^3\) is the process of modifying the source code to a functionality identical version which minimized the size of the code through the removal of whitespace, newlines, and comments as well as the shorting of identifier names. As we detail in our discussion on real world data in chapter 4, and [21] has similarly found, code obfuscation presents a significant challenge to a static analysis. Minification does not pose a significant challenge.

3. Contribution

In the course of this thesis, we compare different browser fingerprinting analyses and their results classifying data from 10,000 websites. In this process we implement a dynamic and lexical analysis for

- Canvas fingerprinting
- Canvas font fingerprinting
- WebGL parameter fingerprinting

\(^3\)https://developer.mozilla.org/en-US/docs/Glossary/minification
```javascript
const obj : Foo = new Foo();

// Computed methods names
const s : string = input();
obj[s]();

// static methods names
obj.method_name_a();
```

Figure 1. A demonstration of computed method names in JavaScript (with type annotations for clarity). After the creation of an object named `obj`, of type `Foo`, the computed method names section shows a method call to `obj` to the method of corresponding to the string `s`. The string `s` is initialized as the return of a hypothetical input function to represent its potentially ambiguous runtime value. Contrarily, the static methods names section shows a method call to `obj` to the method named `method_name_a`. There is no ambiguity as to which method is being called in a static method call.

in chapter 4. We introduce each analysis in enough detail to understand how analysis is using the input data to arrive at a conclusion, without getting caught up in implementation details. 4 With the expectation that no analysis is perfect, we discuss this successes and shortcomings to an analysis, with a focus on the disagreements the dynamic and lexical analyses have. We present quantitative results on how the analyses classified large portions of real world data. Examples of misclassifications the different analysis make on real world data are provided. We hope the detail of shortcomings and mistakes made by current analyses may motivate future iterations of analyses. Durning

4The implementation can be obtained at our publicly available code repository here: https://github.com/tim-stephenson/OpenWPM-Data-Analytics
our analysis of Canvas fingerprinting, we detail our contribution in iterating upon a lexical analysis through unescpaing string literals, based on our observation of widespread character escaping being performed.

4. Motivation for comparison of browser fingerprinting analyses

We compare two different analysis, a lexical analysis which uses static data as input, and a dynamic analysis which uses dynamic data as input. Our motivation for comparison is two-fold. First, comparing different analysis aids in the process in finding false negatives and false positives, as the disagreements between different analyses are entirely the false negatives or false positives of one of the analysis. Secondly, the two analysis we are comparing use different input data (static versus dynamic) and thus cannot always be swapped in place. In the problem of mitigating browser fingerprinting, discussed in chapter 2.6, dynamic data cannot be obtained before executing the scripts, which defeats the purpose of mitigation.
Background on Browser Fingerprinting

1. History of Browser Fingerprinting

1.1. Discovery. Mayer’s senior thesis from 2009 ([26]) uncovered the breadth of information the web browser exposes to a web server through client side scripting. He hypothesizes that this information, which we know as the browser fingerprint\(^1\) may potentially be sufficient to uniquely identify users. The small scale study as part of the thesis found 1278 out of 1328 clients to have unique browser fingerprints.

1.2. Evolution. Browser fingerprinting has been constantly changing throughout its 14 year history. The interface available to client side scripts has evolved from Java Applets, Action Script (what ran Adobe Flash Player) and JavaScript, to just JavaScript today. HTTP headers have also played a consistent role in providing information for the browser fingerprint. Perhaps Web Assembly will exist in future discussions, however today it’s only access to the entropy sources known about today exists through an interface with JavaScript. What the browser fingerprint consists of has been volatile. The list of browser plugins, once a high entropy source, has been deprecated.[16]\(^2\) The Canvas and WebGL APIs emerged as sources of significant entropy. Additionally, browser fingerprinting has increasingly received more serious attention from powerful organizations such as the World Wide Web Consortium and the owners of popular web browsers.

2. The Uniqueness of a User’s Browser Fingerprint

Important to the discussion of browser fingerprinting, is how unique a browser fingerprint is amongst users. Additionally, which parts of the browser fingerprint are

\(^1\)The term browser fingerprinting was not coined yet at the time of Mayer’s thesis.
\(^2\)https://developer.mozilla.org/en-US/docs/Web/API/Navigator/plugins
attributable to the uniqueness is important in determining the more and less effective methods of browser fingerprinting. In order to conclude on the uniqueness, large-scale studies have been performed which extract the browser fingerprint of a numerous users. They present the effectiveness of browser fingerprinting in terms of the entropy given off by a browser fingerprint.

2.2. Entropy as a quantitative measurement of the information in a browser fingerprint. Entropy and normalized entropy provide a rough idea as to the degree to which browser fingerprints differ between users. Entropy conveys how many bits information the average fingerprint gives off. Normalized entropy conveys how many bits of information the average fingerprint gives off as a ratio to the bits of information needed for every fingerprint to be unique in the distribution. As singular quantity, it cannot convey to full story of a distribution, and the studies referenced go further into depth describing their fingerprint distributions. However, we find entropy to provide the best measurement for comparison between different studies and types of fingerprints.

\[ H(X) = \log_2(N) - E\log_2(\text{occurrences of } x) \]

This naturally demonstrates the maximum entropy \( (H_M) \) is \( \log_2(N) \), where every value \( x \) in the distribution is unique, and thus has one occurrence. The maximum entropy is used to compute the normalized entropy of \( \frac{H(X)}{H_M} \).

\[ \frac{H(X)}{H_M} \]

\[ 3 \text{A probability distribution derived from observed data.} \]
2. BACKGROUND ON BROWSER FINGERPRINTING

2.3. Studies on the distributions of browser fingerprints. Three studies have been performed gathering a large-scale, real world distribution of browser fingerprints.

- How Unique Is Your Web Browser? [10]

- Beauty and the Beast: Diverting modern web browsers to build unique browser fingerprints [24]

- Hiding in the Crowd: an Analysis of the Effectiveness of Browser Fingerprinting at Large Scale [16]
  - Preformed in 2018. Collected fingerprints from visitors to a popular French news website.

2.4. Independence between distributions of different types of browser fingerprints. Two distributions are independent if knowing the value of one of the distributions does not affect the probability distribution of the other. For many properties in the browser fingerprint, there is a well-known lack of independence between their probability distributions. An example of pieces of the browser fingerprint not independent are timezone and language. The impact a specific type of browser fingerprinting provides largely has to do with its entropy and independence to other types of browser fingerprinting. If two distributions are independent, then entropy can be naively added to find the entropy of those distributions combined. As they become less independent, their combined entropy decreases until one of the distributions is perfectly dependent on the other, at which point the entropy is the maximum of the original two distributions. The three studies we rely on for our entropy data do not detail the independence between various parts of the browser fingerprint. However, it is important to be aware of the potential limited impact of a new form of browser fingerprinting in the case it is
highly dependent on a previously known variable which is apart of the browser fingerprint. We hope future work on the independence between types of browser fingerprints is preformed.

3. Browser Fingerprinting Types / Methods of Browser Fingerprinting

3.1. Overview. The following section aims to give a broad overview of the many components which may make up a user’s browser fingerprint. Canvas fingerprinting, font fingerprinting, and WebGL parameter fingerprinting are discussed in greater detail as these three fingerprinting types are that which we explore the analysis off in chapter 4.

3.2. Canvas fingerprinting. The HTML Canvas element provides support for drawing shapes, lines, or text, in different sizes and colors onto a mutable image. The pixel data of the Canvas element can be extracted after such modifications are made. It was first discovered that the same instructions to the Canvas can produce differences in resulting pixel data by Mowery and Shacham in Pixel Perfect: Fingerprinting Canvas in HTML5 [28]. They found that differences in the pixel values rendered to a Canvas could be attributed to operating system (due to text kerning), browser version, graphics card, installed fonts, sub-pixel hinting, and anti-aliasing differences. The two main APIs for drawing are CanvasRenderingContext2D (for simple 2D shapes, lines, and text) and WebGLRenderingContext 4 (for complex 2D and 3D renders) Similar to other forms of fingerprinting, Canvas fingerprinting is not obtrusive to the user, as the Canvas element generated does not need to be put onto the DOM for the user to view.

We look at CanvasRenderingContext2D because of its greater prevalence on the web and academic work on the topic. It terms of prevalence of usage, out of 7875 functional websites we crawled, 2108 used CanvasRenderingContext2D, whereas only 45 used WebGLRenderingContext5.

4 Alternatively WebGL2RenderingContext could be used. WebGL2RenderingContext conforms closely to OpenGL ES 3.0. WebGLRenderingContext conforms closely to OpenGL ES 2.0.

5 This refers to use of the respective APIs at runtime. As we later investigate, there is a substantial amount of scripts which include WebGL code that did not execute.
Prior academic studies on Canvas fingerprinting have investigated its effectiveness. These studies which embedded Canvas fingerprinting functionality into websites, give good estimate about how effective Canvas fingerprinting is at identifying users. The two studies, whose results are summarized in Table 1, agree in their findings that a Canvas fingerprint provides about 8 bits of entropy on the user. This relatively high entropy available in the Canvas fingerprint motivates further study of Canvas fingerprinting.

Prior studies on Canvas fingerprinting have investigated its prevalence. Their findings are listed in Table 2.

3.3. Font fingerprinting. The goal of font fingerprinting is to gather the list of font the user’s machine supports. This list of supported fonts makes up the font
fingerprint. There are many different methods used to achieve this, such as the Canvas API, a HTML span element, or `window.queryLocalFonts()`\(^6\). The `window.queryLocalFonts()` function, the easiest method, is not used due to its low browser support and explicit user permission required. Using the Canvas API or a HTML span element to detect installed fonts is a instances of exploiting functionality quirks, as it requires repeatably rendering text with a large number of different fonts, then testing the success of such render. When the Canvas API is the interface used for font fingerprinting, it may be referred to as Canvas font fingerprinting.

The methodology to perform font fingerprinting via either a HTML span element or the Canvas API is similar. Both try rendering the same piece of text with many fonts, and measure the text properties (typically the width) of each render to determine whether the font was properly rendered, or a fallback/control/default font was used (implying the user does not support a given font).

The method of Canvas Font fingerprinting we look for utilizes the `CanvasRenderingContext2D.font` property to change the font to that whose presence is being queried. Then the `CanvasRenderingContext2D.measureText()` method is used to determine if the font results in rendering text with dimensions distinct from the fallback font.

Though we do not analyze font fingerprinting which utilizes the HTML span element, it is a popular form of font fingerprinting. Open source FingerprintJS v3.4.0, uses the HTML span element with the same abstract design as we have describe above.\(^7\) Additionally, one could imagine a different Canvas Font fingerprinting design which renders the text with `fillText`, and then extracts the render with `toDataURL` or `getImageData`. These renders could then be compared just as the dimensions returned

---
\(^6\)`window.queryLocalFonts()`, a part of the Local Font Access API is detailed in the ‘unofficial draft’ at [https://wicg.github.io/local-font-access#font-manager-api](https://wicg.github.io/local-font-access#font-manager-api)

\(^7\)`Implementation fixed to current version: https://github.com/fingerprintjs/fingerprintjs/blob/v3.4.0/src/sources/fonts.ts`
2. BACKGROUND ON BROWSER FINGERPRINTING

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>$H$</strong></td>
<td>13.900</td>
<td>8.379</td>
<td>6.904</td>
</tr>
<tr>
<td>Normalized Entropy</td>
<td>0.738</td>
<td>0.497</td>
<td>0.329</td>
</tr>
<tr>
<td>Study size</td>
<td>470,161</td>
<td>118,934</td>
<td>2,067,942</td>
</tr>
</tbody>
</table>

**Table 3.** The entropy ($H$) for the list of fonts found by Panopticlick [10] AmIUnique [24] and Hiding in the Crowd [16]

<table>
<thead>
<tr>
<th>Rank Interval</th>
<th>1-million-site Measurement (2016)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1K)</td>
<td>2.50%</td>
</tr>
<tr>
<td>[1K,10K)</td>
<td>1.98%</td>
</tr>
<tr>
<td>[10K,100K)</td>
<td>0.86%</td>
</tr>
</tbody>
</table>

**Table 4.** The prevalence of websites implementing Canvas font fingerprinting found by 1-million-site Measurement (2016) [11]

by `measureText` are compared to determine the installed fonts. This would bypass usage of the `measureText` method.

Despite the many ways of gathering the available fonts, if they are accurately obtaining the installed fonts, they are equally useful regardless of method. The usefulness of a fingerprint is well described by its entropy value. Three studies have looked at the entropy that the list of installed fonts provides. These are listed in Table 3. Even the lowest found bits of entropy still find near a byte of identifying data is available in the list of installed fonts.

A Prior study on Canvas fingerprinting have investigated its prevalence. Their findings are listed in Table 4.
3.4. **WebGL parameter fingerprinting.** The WebGL API provides access to parameters related to the underlying graphics rendering, in order to aid in a complex rendering task. These parameters can be queried passing the `WebGLRenderingContext.getParameter` method the enum value corresponding to the parameter of interest. For some parameters, a WebGL extension must first be enabled to querying the parameters associated with the extension. Specifically, the `UNMASKED_VENDOR_WEBGL` and `UNMASKED_RENDERER_WEBGL` parameters, apart of the `WEBGL_debug_renderer_info` extension, provide a good fingerprinting target. These parameters have been used in fingerprinting studies, such as [22] and [42]. Additionally, open source FingerprintJS v3.4.0 uses these two parameters. These return strings corresponding with the vendor of the underlying graphics driver (e.g. the company which made the drivers) and the renderer of the of the underlying graphics driver (e.g. what GPU and driver version the user is using). Some examples of `UNMASKED_VENDOR_WEBGL` and `UNMASKED_RENDERER_WEBGL` values are shown in Figure 1. An example of how to query these parameters in JavaScript is shown in Figure 2.

In terms of stability over time, the physical GPU is a stable property of a user’s machine. The physical GPU is often exposed in the `UNMASKED_RENDERER_WEBGL` value. The Entropy from the WebGL vendor and renderer from a variety of studies are given in Table 5.

3.5. **WebRTC fingerprinting.** The Web Real Time Connection (WebRTC) API allows for peer to peer connections between web clients. WebRTC fingerprinting retrieves the public and private IP addresses of the client through the WebRTC API. The WebRTC API’s ability to leak the public and private IP address of a user, has been detailed in [33]. The WebRTC fingerprint is thus made up of the public and local IP addresses of the client.

---

8These are the two WebGL parameters which we will analyze scripts on whether they are querying those parameters. There are likely are other WebGL parameters which provide a good fingerprinting target. We do not investigate any of these other parameters, however, we do observe many fingerprinting scripts querying parameters beyond `UNMASKED_VENDOR_WEBGL` and `UNMASKED_RENDERER_WEBGL`. 
Figure 1. Example values for WebGL Vendor and WebGL renderer provided found in [22]

```javascript
const canvas = window.document.createElement("canvas");
const gl = canvas.getContext("webgl");

const debugInfo =
    gl.getExtension("WEBGL_debug_renderer_info");
console.log(
    {
        "UNMASKED_VENDOR_WEBGL":
            gl.getParameter(debugInfo.UNMASKED_VENDOR_WEBGL),
        "UNMASKED_RENDERER_WEBGL":
            gl.getParameter(debugInfo.UNMASKED_RENDERER_WEBGL)
    });
```

Figure 2. example of how to query the UNMASKED_VENDOR_WEBGL and UNMASKED_RENDERER_WEBGL WebGL parameters.
Table 5. The entropy ($H$) for the WebGL vendor and renderer found by (Cross-)Browser Fingerprinting [42] AmIUnique [24] and Hiding in the Crowd [16]

3.6. Information querying. A large number of differential attributes of a web user do not require a complex scheme to gather, but rather can be freely queried from the Web API. A non-exhaustive list of forms of information querying are as follows:

- CSS media features
- Navigator properties
- WebGL parameters
- Permissions
- Screen properties
- Timezone

4. Applications of Browser Fingerprinting

Browser Fingerprinting’s primary usages from a website owners’s perspective, are user tracking and augmented authentication.

4.1. User tracking. The browser fingerprint, as a fingerprint, may be attempted to be used to identify and track users between different sessions and websites. The most common appeal of user tracking is to build a profile on a user, to then serve them
targeted advertisements. This is achieved through browser fingerprinting, by a advertisement network putting their third-party fingerprinting scripts throughout the web. Consequently, as long as the user’s fingerprint is unique and stable, the advertisement network would have a profile on the user’s web browsing habits, for which they can then target advertisements. This type of tracking is called ‘stateless tracking’. Stateless tracking is a form of tracking which does not require a state (often a cookie) stored on the users device, the opposite of to ‘stateful tracking’. Stateful tracking, through the likes of third-party cookies[15] is the primary methods for tracking users. However third party cookies are being phased out[10] leading to increased interest in browser fingerprinting as an alternative user tracking methodology. Additionally browser fingerprinting can be used for cookie regeneration, a hybrid of stateful and stateless tracking that uses the browser fingerprint to regenerate the user’s state in the event it was deleted/cleared.[10][14][11]

4.2. Augmented authentication. The information the browser fingerprint reveals about the client may be used to detect and prevent misuse of a computer system with web interaction. For example, consistency in the browser fingerprint can be used to determine the authenticity of a user, such as in done in Picasso [6] below. Alternatively, the browser fingerprint can be added to a anti-fraud system which attempt to determine user variables which correlate with fraud. When the browser fingerprint is used to as a supplementary security check to prevent unauthorized access, we say it is being used to augment authentication. Theoretically, the browser fingerprint, which provides a plethora of information, may be useful in making a determination of whether a user logging in is a legitimate authorized account holder, or a fraudster. Companies such as

[9]When interacting with a website, a third-party is any webpage server from a different domain than that of website being visited. The website being visited is the first-party. Third party cookies are thus cookies from a third-party.


[11]This would be useful for a website attempting to evading tracking protections, such as Safari’s 7 day third party cookie lifespan https://github.com/WebKit/WebKit/pull/5347
Imperva [20], LexisNexis Risk Solutions [37] Human [19], and FingerprintJS Inc. [12] all market services which utilize browser fingerprinting for (claimed) security purposes. A typically browser fingerprint augmented authentication workflow goes as follows:

1. Collect the browser fingerprint of a new device logging in to a user’s account, or creating a new account.
2. Upon re-login, if a devices browser fingerprint does not match a previously authenticated device, trigger some additional form of authentication (2FA).
3. After a certain session length, or upon a certain trigger during a authenticated session (such as IP address change) re-check the devices browser fingerprint, if the browser fingerprint does not match the browser fingerprint during session creation, invalidate the session. This may help prevent session hijacking attacks.

A challenging aspect of using the browser fingerprint for security purposes is dealing with actors which may spoof parts of the fingerprint.

A team at Google published their system for using browser fingerprinting to augment authentication in the paper Picasso: Lightweight Device Class Fingerprinting for Web Clients [6]. They used Canvas fingerprinting to verify the integrity of a client’s operating system and browser. They were able to determine the client’s legitimate operating system and browser from the Canvas fingerprint with 100% accuracy. They would compare the OS and browser information in the user agent string to that determined by their Canvas fingerprinting machinery. Doing so would detect clients spoofing their device information. They found the detection of clients spoofing their device information useful in detecting inorganic interactions from attackers emulating their device for nefarious purposes.

5. Prior work on analysis/detection of Browser Fingerprinting

5.1. Overview. Often privacy studies which look for characteristics such as fingerprinting also look for other privacy invasive actions such as stateful tracking. Because
we are looking for detection methodology’s of browser fingerprinting, we only discuss the portions relevant towards this goal.

Most previous research which implements a methodology for detecting browser fingerprinting roughly follows the following format.

1. Detail their methodology for detecting browser fingerprinting.
2. Run a web crawl which gathers the data needed for input in their analysis mechanism.
3. Present the results from their analysis of browser fingerprinting prevalence in their web crawl.

Of importance is a browser fingerprinting analysis depends on specific types of input data gathered during a web crawl.\textsuperscript{12} As previously discussed, the two types of data which can be gathered during a web crawl are dynamic and static. We detail the work done towards analysis of browser fingerprinting using dynamic data and static data bellow\textsuperscript{13}.

5.2. Detection through dynamic data. To our knowledge, the first study to instrument aspects if the web browsing experience to detect browser fingerprinting was [1]. They instrumented portions of the Canvas API associated with Canvas fingerprinting. They used heuristics on the dynamically instrumented data to determine which script were preforming Canvas fingerprinting. [11] also performed a dynamic analysis based on instrumentation to interactions with various Web API’s. They build on [1] to come up with slightly altered heuristics for Canvas fingerprinting, to take into account changes in how sites implemented Canvas fingerprinting. They did not describe (or give an implementation) to how this was achieved. However, they did provide their web crawler source code, OpenWPM, which details the format for their dynamic data (and is actively maintained). The method of translating their requirements into an

\textsuperscript{12}The data does not necessarily need to be gathered during a web crawl. For example, it could be apart of some background web extension which gathers data during while a human browses the web.

\textsuperscript{13}This is (obviously) a non-exhaustive list studies which worked towards a analysis/detection of browser fingerprinting.
implementation is straightforward, however it is short of details such as the potential for multiple Canvas element instances. [11] continues with heuristics for Canvas font fingerprinting and WebRTC fingerprinting based on dynamic data.

[21] similarly uses OpenWPM to instrument dynamic data pertaining to WebAPI’s the authors deemed important for fingerprinting. They use a machine learning approach, where the dynamic features they derive from their dynamic data are as follows:

- Count of the number of times a individual script accesses each individual API method or property of interest.
- For specific aspects of the Web API which may have values set/accessed, methods called with arguments, or methods return values, they create a higher-level features which do not include the value, arguments, or return value themselves, but some derived property from them. An example\textsuperscript{14} of this they gave was:
  - The features corresponding to a \texttt{CanvasRenderingContext2D.fillText()} are the area of the corresponding canvas element, the length of the string passed as an argument, and whether the canvas element is present on the user’s screen\textsuperscript{15}

With this feature set they follow a machine learning process of:

1. Run large scale crawl (using 20k domains)
2. Establish ground truth using (dynamic analysis based) heuristics nearly identical to [11]
3. Filter out features with variance < 0.01

\textsuperscript{14}They did not give a complete list of what their derived features were, only examples.
\textsuperscript{15}To the best of our knowledge they did not explicitly describe how they determined the area of a canvas element or if it is present on the screen (i.e. it is present in the DOM). This dynamic data is not available through any configuration setting of OpenWPM. They published the extended version of OpenWPM they were using on github, but not any part of their feature extraction process from OpenWPM resulting crawl data. They mention that they were instrumenting DOM interaction (through \texttt{createElement, document, node}) which is perhaps part of the methodology used.
(4) Filter the top 1,000 features by information gain\cite{32}\textsuperscript{16} computed using the initial heuristics based ground truth

(5) Train a decision tree using the current ground truth. Classify the data using the this decision tree, and manually analyze disagreements between the previous ground truth and the classifier’s output to re-define the ground truth.

(6) Perform three total iterations of step 5

5.3. Detection through static data. \cite{1} discusses the use of static analysis to detect extraction of plugins, navigator properties, screen properties. We believe their static analysis is a manual (human) analysis of the source code.

\cite{40} used a lexical analysis to gather counts of specific calls to pieces of the Web API. They use a pre-processing step of de-obfuscation and expanding member expression before performing their lexical analysis. They used the resulting counts from their lexical analysis as the features in their support vector machine they used to classify scripts as fingerprinting. The keywords they searched for in their lexical analysis step were, some \texttt{window.navigator}\textsuperscript{17} properties, some \texttt{window.screen} properties, and \texttt{Date.getTimezoneOffset}. Their de-obfuscation step contained using js-beautify.\textsuperscript{18} This source code modifier primarily inserts spaces and newlines in a methodological manner to improve readability. Additionally it can decode escaped string literals.\textsuperscript{19} However no mention of the desirability of that functionality was made by the authors.

Their ‘expanding member expression’ step consisted of walking through a AST representation of the source code, keeping track of variables which are assigned to other identifiers in the current scope. Then, in the occurrence of a MemberExpression, they

\textsuperscript{16}The information gain that a feature provides is the amount of information a gained about the target variable/label (in this case, whether or not a script is fingerprinting) given a observation of a feature. Unlike the variance of a feature, this requires labeled training data to compute.

\textsuperscript{17}Though for an unknown reason, they drop the \texttt{window}. prefix

\textsuperscript{18}\url{https://www.npmjs.com/package/js-beautify}

\textsuperscript{19}Chapter 4.2 discusses the benefit of escaping strings in a lexical analysis.
2. BACKGROUND ON BROWSER FINGERPRINTING

```javascript
// source code
var nav=navigator;

function fingerprint(){
    var a = nav.plugins;
    var b = a;
    var c = b.length;
    var d = nav.userAgent;
}

// after expanding member expressions
var nav=navigator;
function fingerprint(){
    var a = navigator.plugins;
    var b = navigator.plugins;
    var c = navigator.plugins.length;
    var d = navigator.userAgent;
}
```

Figure 3. Example from [40] of how a lexical search for access to the Web API (`navigator.plugins.length`, `navigator.userAgent`) can fail when not directly referenced. This motivates their methodology to expand method expression in the source code.)

attempt to replace the object with it’s previously assigned value in the current scope.\(^\text{20}\)

There intent in doing so is to circumvent code such as in Figure 3 not being picked up by a lexical analysis looking for keywords such as `navigator.userAgent`.

Just as [21] defined dynamic features, and went through a machine learning process with those features and real world data as described in subsection 5.2, they went

\(^{20}\)Read more about the JavaScript AST format here: https://github.com/estree/estree
through a similar process with features derived from the static data. First they instrumented both external and inline JavaScript source code during their web crawl using OpenWPM. Then they had a pre-processing step for the scripts which ‘unpacks’ the portions of the script which contains a `Function()` or `eval()` call to a string literals. `Function()` and `eval()` allows for arbitrary JavaScript code in string form to be executed. They unpacked JavaScript source code which was is in the form of a string literal ‘packed’ into a `Function()` and `eval()` in order to obtain a more representative AST of the actions the script performed. For instance, if the entire script was packed into a `Function()` or `eval()`, the AST representation script would entirely be one ‘CallExpression’ with a ‘StringLiteral’ as the sole argument, thus not very useful. After this pre-processing step, the static features they went about extracting features from the source code for their ML model. Their choice of features were as follows:

- A pair of a parent node, and direct child is a feature. Note that this is storing more information than the type of parent and child node. For example, a literal, or identifier additionally store their value and keyword respectively within the AST node.
- Only parent node, child node pairs where at least one of the nodes, contains at least one keyword which matches a name, method, or property from any JavaScript Web API.\(^2\)
- The feature is a binary feature. The existence of the unique parent node, child node pair in a script gave the feature a value of 1, 0 otherwise. Thus multiple occurrences of the same parent node, child node pair where treated equally to a single occurrence.

\(^2\)They did not explicitly mention a direct child, however the examples they gave where all of direct child nodes, and the term child node typically means direct.

\(^2\)The methodology of aggregating all of the keywords associated with all of the JavaScript Web API’s was not given. They linked to the Mozilla MDN Web docs as their reference for finding all of the keywords. [https://developer.mozilla.org/en-US/docs/Web/API](https://developer.mozilla.org/en-US/docs/Web/API)
They then followed the same machine learning steps as they did with the dynamic features. Notable in this process is they start with a heuristics based dynamic analysis as the ground truth to prune static features, and then iterate their ground truth based on the static features. Additionally they found 6.4% of scripts failed to be parsed into a AST.

6. Mitigation of Browser Fingerprinting

6.1. Overview. There are many existing consumer tools/products available with the intent of decreasing the likelihood that a user can be tracked via their browser fingerprint on the web. The following categories of fingerprinting mitigation strategies are used:

- Block API access
  - An extreme degree of blocking API access is the disabling JavaScript.
  - A less extreme example is Firefox’s experimental ‘Fingerprinting Protection’ which block’s access to the Canvas API unless a user gives explicit permission (among other blocked/disabled features).

- Override API’s to send uniform values
  - Sometimes referred to as the ‘Tor approach’, this methodology overrides the Web API to make a clients configuration and functionality as similar to a ‘typical user’ as possible, thus blending in among a common browser fingerprint shared among many users.

- Override API’s to send randomized values
  - This methodology aims to make a users browser fingerprint unstable, by randomizing aspects of the Web API used for browser fingerprinting. An example of this is adding ‘noise’ to Canvas API renders in a randomized fashion, as done by [23].

23 As of Firefox v112.

If a user’s browser fingerprint is unstable enough such that two extractions of their browser fingerprint yield browser fingerprints which are too different to be grouped together, then that user cannot be tracked via browser fingerprinting.

- Filter requests from a ‘filter list’ of known fingerprinters
  - Additionally, a filter list can be used in coordination with any of the previous three strategies to conditionally apply a mitigation strategy based on if a script is apart of a filter list of known fingerprinting scripts.
  - This process is much more common among the mainstream/popular fingerprinting mitigators. For example, the latest version of Firefox uses ‘Enhanced Tracking Protection’ by default to block network requests to known fingerprinting scripts.\(^{24}\)

\(^{24}\) As of Firefox v112.


Note the ‘standard’ setting to ‘Enhanced Tracking Protection’ is enabled by default, which claims to filter out known fingerprinters.
CHAPTER 3

Real World Website Data Gathering

1. Overview

In section 2 we discuss the basics of a web crawler, why we used one to gather our data, and the OpenWPM crawler we used. In section 3, we discuss our usage of OpenWPM to collect our data. In section 4, we discuss the flaws in the data we gathered, as well as possible alternatives that could be pursued in future work.

2. OpenWPM

A bot which systematically browses the web is useful in many circumstances, such as search indexing, site scraping, performing a web census, etc. A web crawler is useful for us to visit a large number of websites and store data associated with their visit. The choice of a automated crawl brings with it the downside of a lack of authentic user interaction with the page, as we will discuss in section 4. However, a automated crawl benefits in simplicity and reproducibility.

We use OpenWPM [11] to perform our web crawl. OpenWPM aids in deployment of a web crawler which can be configured to save a variety of data important to privacy researchers from the websites it visited. OpenWPM controls a Selenium web driver. The Selenium web driver controls a fully-fledged Firefox browser with extensions to instrument parts of the web browsing experience. The web browser which Selenium controls can be configured to be headless, headful (native display), or headful with the X virtual frame buffer (xvfb) used as a display.¹ The data gathered from these browser

¹Xvfb is an X server designed to operate on computers without display hardware or physical input devices. It simulates a basic framebuffer using virtual memory. https://www.x.org/releases/X11R7.6/doc/man/man1/Xvfb.1.xhtml
extensions gets sent back the OpenWPM program which acts as a data aggregator. It can store a range of information from the web crawl, such as run-time javascript actions, network requests, and stateful measurements (e.g. cookies). The data aggregator sends the data to the users choice of structured and unstructured storage provider. The options for structured data are Parquet and Sqlite. The options for unstructured data are LevelDB and native file system storage. Additional storage options are possible by implementing OpenWPM’s data aggregator class.

3. Parameters used in OpenWPM

3.1. Sites crawled. We used a medium sized web crawl of the top 10,000 websites on January 31st, 2023 according to research-oriented top sites ranking Tranco[25]. Though we attempted to connect to 10,000 website, only 7,875 were functional. We deemed a website to be functional if the web server responded to at least one HTTP network request websites with a 2xx success response code. The 2,125 non-functional website failed to successfully responded to the HTTP request from their HTML body. Attempting to connect to a handful of these non-functional websites on our personal computer repeats the failed connection which occurred during the web crawl.

3.2. Headless versus headful. We used a headless browser to preform our crawl. The decision not to preform a headful crawl was the result of choosing the path of least resistance as we were working with a Debian server without native display capablites, and a failure to configure xvfb. Though we did not perform a performance comparison, headless crawling will have a performance benefit from avoiding rendering anything. We discuss the negatives of a headless browser in subsection 4.2

3.3. Persistent state versus cleared state. OpenWPM allows the option for either a state (e.g. cookies, local storage) which persists between connections to different websites, or a state which is cleared between connections to different websites. We

\[2\] Available at https://tranco-list.eu/list/Q9KZ4. The first 10,000 by ranking were used.
choose to clear the state between connections to different websites so that the visit to each website is independent of one another.

3.4. Data instrumented. In order perform the different analyses desired on the source code and the run-time data, both needed to be saved. HTTP requests were instrumented to save the scripts sent over a network request. Dynamic JavaScript instrumentation was enabled. The settings for which portions of API to instrument are detailed in Appendix A.

4. Limitations in our data gathering using OpenWPM

4.1. Only firefox browser. As previously discussed, the implementation of the Web API is standardized by the World Wide Web Consortium, but implemented at the discretion of the web browsers. Currently, there is often a significant difference between different web browser’s conformance to the Web API, especially among newer and experimental portions. For example, the Battery Status API is not available in Firefox, but is available in Chrome. The limitation to only the Firefox browser prevents any analysis that pertains to an API Firefox does not support or has limited support for.

4.2. Headless browser. In using a headless browser, it presents the potential that websites may block or alter functionality based on the detection of a headless browser. A website may detect the use of a headless browser through the

window.navigator.webdriver

property. We observed many sites accessing this property. Additionally we observed many scripts which referenced ‘Selenium’ and ‘web crawler’ which we would guess are detecting the presence of a web crawler.

4.3. No inline JavaScript saved. As discussed in section 3, a script’s source code is saved when sent over a network request. This misses the scripts which are inline JavaScript. Inline JavaScript is JavaScript code, in the body of a HTML script element, shown in Figure 1.
3. REAL WORLD WEBSITE DATA GATHERING

```javascript
// external script
<script type = "text/javascript"
src = "https://www.example.com/example.js"></script>

// inline script
<script type = "text/javascript">
  ...
</script>
```

**Figure 1.** A demonstration of the differences between external JavaScript and inline JavaScript.

This can be overcome in a variety of ways, such as saving the HTML and then extracting the inline JavaScript. Though they did not publish their code on how they did so, [21] did accomplish this. This was an engineering challenge not prioritized, and thus was not accomplished.

Because of the lack of inline JavaScript, there are scripts for which the dynamic data was properly instrumented, but no source code was saved. No comparison can be made between an analysis which uses dynamic data and one that uses static data, if the source code is not available.

**4.4. No page interaction.** Our web crawl visits the home page’s of 10,000 websites. This is potentially missing a large amount of the functionality of the web which is in sub-pages or the result of user interaction (e.g. user login). Of the previous prevalence studies done on the fingerprinting, [30] and [2] have looked at the home page, as well as sub pages, either by clicking links on the homepage or parsing the sitemap.xml websites provide. However, many other studies crawl only the homepages, such as [1], [4], [11], and [21]. A homepage only crawl may miss functionality that occurs during

---

3A website’s sitemap, an xml file often stored at https://www.example.org/sitemap.xml is a format which details the webpages available for search engine indexing.
and after user logins. As our discussion in chapter 2.4 discusses, the two primary uses for browser fingerprinting are user tracking and augmented authentication. Web crawls which do not visit login-in or other pages with security implications may miss augmented authentication functionality occurring during these pages. However, as user tracking would most likely occur upon page load for a user, we believe that a homepage crawl should be sufficient for finding browser fingerprinting deployed for the purpose of user tracking. Additionally, as user tracking is the more nefarious application of browser fingerprinting, we are more interested in finding instances of user tracking than augmented authentication.\footnote{In our research and discussion of found browser fingerprinting scripts in chapter 4, we did not attempt to classify occurrences of fingerprinting in terms of augmented authentication or user tracking. We do discuss fingerprinting scripts which come from companies providing browser fingerprinting services for the purpose of (supposedly) augmented authentication. However, from the web client’s perspective, the purpose of the extraction of the browser fingerprint is unknown beyond guesses based on the context of the website.}
CHAPTER 4

Analysis Method and Results

1. Overview

In the following three sections we follow a similar pattern for detailing our analyses and subsequent results of these analyses. The structure is roughly the follow steps

(1) Detail a high level understanding of what behavior constitutes performing the respective type of fingerprinting.

(2) Detail a translation from our high level requirements to our dynamic and lexical analysis, including how it may fall short of perfectly implementing our high level requirements.

(3) Present the results of our analysis on our ‘domain’ from our web crawl. We used a domain of the cartesian product between websites (specified by their domain name) and scripts (specified by the URL they came from). Thus a single element in our input domain, is a site, script pair. We will also refer to these elements as site, script pairs. Note that identical scripts may be delivered from many different script URL’s and that the same script URL may return many distinct scripts. In the case that a site, script pair corresponds to multiple saved scripts, if any of the scripts are classified as fingerprinting, we consider the site, script pair to be fingerprinting.

(4) Detail characteristics of resulting site, script pairs which were classified as being in the following buckets:
   • Classified as fingerprinting by both the dynamic and lexical analysis.
   • Classified as fingerprinting by the dynamic analysis, but not the lexical analysis.
   • Classified as fingerprinting by the lexical analysis, but not the dynamic analysis.

\[\text{33}\]
• Classified as fingerprinting by the lexical analysis, and no dynamic data was available.

This includes a manual assessment of if the site, script pairs were truthfully fingerprinting, and a discussion as to the explications for a analyses misclassification when relevant. We limit our scope to manually investigating site, script pairs which were classified as fingerprinting by at least one of our analyses, as this group is relatively small (around 1,000-3,000) compared to all other elements (around 250,000).2

2. Canvas Fingerprinting

2.1. Characteristics of canvas fingerprinting in static and dynamic crawl data. Canvas fingerprinting is an instance of exploiting functionality quirks. The use of CanvasRenderingContext2D, for drawing graphical content onto the users page(i.e. its legitimate purpose) is difficult to distinguish from websites using CanvasRenderingContext2D for Canvas fingerprinting.

We start by using the following specification of Canvas fingerprinting, a slight adjustment from the specification used by [11]

(1) The canvas element’s size is greater than 16px x 16px
(2) Text must be written to canvas with least 10 distinct characters, using either the fillText or strokeText.
(3) The script extracts an image with a call to toDataURL or getImageData that specifies an area with a minimum size of 16px x 16px.

These requirements are the minimum a script must execute to perform browser fingerprinting. There are many other modifications the CanvasRenderingContext2D can

2These number of elements are for our web crawl which attempted to visit 10,000 websites.
perform to a Canvas element beyond writing text. In every publicly available fingerprinting source code we found, at least\(^3\) writing text to the Canvas was performed.\([13, 38, 42, 5]\)

Implementing this specification as a dynamic analysis is straightforward, as every action defined above is instrumented and saved to the dynamic data. However, as part of dynamic data available through OpenWPM, there is no way to differentiate instances of the same class. For example, the data provided does not give insight as to which instance of CanvasRenderingContext2D is making a call to the toDataURL method, just that a call to toDataURL is being made by some CanvasRenderingContext2D object. Thus if a script manipulates many instances of CanvasRenderingContext2D, the dynamic data can not conclude on whether a Canvas which is being extracted by toDataURL, had a prior method call to fillText. We leave this problem of correlating Canvas API method calls to specific CanvasRenderingContext2D instances open to future research.

We conduct our analysis under the potentially incorrect assumption that all method calls are made to the same Canvas element.

This inability of tracking information flows in this dynamic tabular data, is attempted to be solved in EssentialFP\([35]\). In their work, they use a dynamic information flow control (IFC). This IFC tags data at runtime with a security label. When one value explicitly or implicitly depends on another value, the security label is passed onto it. They use this information flow to detect when Web API sources reach network sinks. We envision using a similar information flow analysis could be used to track the modifications made to a Canvas element, up until a call to toDataURL or getImageData.

Our resulting dynamic analysis are the following conditions:

(1) \texttt{CanvasRenderingContext2D.fillText} was called with a string of at least 10 distinct characters.\(^4\)

\(^3\)Many scripts would have a more complex methodology, which drew other items as well as text, such as shapes or lines.

\(^4\)Notably, calls to \texttt{CanvasRenderingContext2D.strokeText} this was not included in our dynamic conditions. This was not a intentional decision, but rather a mistake.
const x = new Foo();
const y = x.toDataURL();

Figure 1. A false positive, when grep is used to find the method call 
CanvasRenderingContext2D.toDataURL()

(2) HTMLCanvasElement.toDataURL was called or 
CanvasRenderingContext2D.getImageData was called, specifying a width and
height with a absolute value of at least 16px.\textsuperscript{5}

(3) The maximum value HTMLCanvasElement.height and
HTMLCanvasElement.width were set to at least 16px. Alternatively, the
value never being set satisfies this condition.\textsuperscript{6}

We also implement these requirements as a lexical analysis. We look for the oc-
currences of method names of interest in the source code. This will not be able to
determine the inputs to the method calls, only their existence. As this analysis is only
looking for the existence of these method calls, we expect false positives from scripts
which are using the method calls for non-fingerprinting uses. Additionally, the method
names being looked for may not be unique to the CanvasRenderingContext2D class,
leading to false positives exemplified by Figure 1\textsuperscript{7}. Lastly, we expect obfuscated code to
present a challenge, as obfuscated JavaScript code is often absent coherent words. This
has presented problems on previous attempts to statically detect fingerprinting. [21]

Our resulting lexical analysis are the following conditions:\textsuperscript{8}

\textsuperscript{5}The width and height can be negative to signify the direction as detailed here:
\url{https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/getImageData}

\textsuperscript{6}The absence of the Canvas element’s height or width being set signifies the default Canvas size
was used, which is 300 pixels by 150 pixels. This default size satisfies our size requirement.

\textsuperscript{7}Of the method names we are searching for, we believe they are unique to their respective Web API
(in this case the CanvasRenderingContext2D class). However, it is possible that a user defined class
shares method names with the method names we are searching for.

\textsuperscript{8}Notably, we do not include the . prefix to our keywords. Typically expect a method call to exist
in the form \texttt{obj.method_name()}. However, because of computed method names, in our exploration we
Table 1. Results of the lexical analysis and dynamic analysis on 7,875 functional websites for Canvas fingerprinting. Note this is over the domain of (website, JavaScript script URL). A result of Yes signifies the analysis classified the site, script pair as fingerprinting. A result of No signifies the analysis classified the site, script pair as not fingerprinting. A result of N/A signifies the analysis did not see any data from the script, site pair. 9

(1) There exists the keyword `fillText` or `strokeText` in the source code
(2) There exists the keyword `toDataURL` or `getImageData` in the source code

2.2. Results. Table 1 presents a numerical summary of the results from the dynamic and lexical analysis.

There were 530 site, script pairs which were classified as Canvas fingerprinting by both the dynamic and lexical analysis. These 530 site, script pairs contained 391 unique scripts. 10 Every script was manually verified, and found to be truthfully performing

---

9 Often found our method names of interest to be exist as string literals in the source code. For example searching for `.fillText` would miss `fillText` as a string literal.

10 For the dynamic analysis, the analysis does not receive any data when the script does not interact with any portions of the Web API being instrumented. For the lexical analysis, the analysis does not receive any data when the script was an inline script and the source code was not saved.

11 A drop off from the number of unique (website, JavaScript script URL) pairs to the number of unique JavaScript scripts is due to different (website, JavaScript script URL) pairs corresponding to exactly the same source code. All source code was successfully saved.
Canvas fingerprinting. Out of all the websites crawled, none satisfied the simple requirements specified, except to perform Canvas Fingerprinting.

The following methodology was used to perform a manual analysis of the scripts. The scripts were formatted using prettier\textsuperscript{11} to enhance readability. The following signs of Canvas fingerprinting were picked up while reading through the scripts, beyond what was specified in our requirements.

- The text being written to the Canvas via \texttt{fillText} is:
  - The perfect pangram "Cwm fjord bank glyphs vext quiz" or some subsequence of it, or some other pangrams.
  - The emoji grinning-face-with-big-eyes, which may be represented as its code point 0x1F603 (128515) or surrogate pair 0xD83D 0xDE03 (5535756835)
  - Not discernibly functional, a seemingly random sequence of characters

- The Canvas element created by \texttt{window.document.createElement("canvas")} is not preserved or added to the DOM.

- The word fingerprinting used in script in comments, variable names, function names, etc.

- Other forms of fingerprinting adjacent to the Canvas fingerprinting.

These signs were used to manually determine if a script was Canvas fingerprinting.

In the process of manually analyzing all of the script in this group, we found most scripts were instances of a handful of companies’ proprietary fingerprinting code, and open source fingerprinting libraries. Differences between scripts that came from the same library were largely due to version differences or differences resulting from a transcompilation\textsuperscript{12} process. The fingerprinting libraries found in order of prevalence are, FingerprintJS \cite{FingerprintJS}, FingerprintJS Pro \cite{FingerprintJS-Pro}, Snowplow \cite{Snowplow}, Adscore \cite{Adscore}, ClientJS

\textsuperscript{11}https://prettier.io/ is a an code formatter primarily used to ensure style consistently

\textsuperscript{12}Transcompilation, being the compilation of source code from high level source code, to another form of high level source code. In the JavaScript world, forms of transcompilation include minification, obfuscated, and module bundlers.
Figure 2. This code fragment was taken from source code classified as Canvas fingerprinting by both the dynamic and static analysis. This segment is part of a 10,000 character long string literal which is then modified and passed into the `eval()` function.

[38], Perimeterx (now HUMAN) [19], Tealium [39], RevOffers [34], E-Hawk [9]. We are able to determine what library/company/service a script came from, as many of the scripts had header comments detailing the licensing information, library name, library version, company name, etc.

These scripts were obfuscating to varying degrees. The obfuscation made it challenging but not impossible to manually investigate if Canvas fingerprinting was being performed. Figure 2 shows one of the more obfuscated scripts in this group. The existence of the phrase "Cwm fjord bank glyphs vext quiz" after `fillText` is a sure tell the script is fingerprinting. Other signs of Canvas fingerprinting include the "ud83d|ude03" surrogate pair for the grinning-face-with-big-eyes emoji and the signs of WebGL parameter fingerprinting (or other types of fingerprinting) nearby in the code.

In the group of scripts classified as fingerprinting by both the lexical and dynamic analysis, we looked for false positives, but found none. As we discuss below, there are
4. ANALYSIS METHOD AND RESULTS

```javascript
var _ = [
    "\x6d\x61\x74\x63\x68",
    "\x31\x2e\x39",
    "\x5c",
    "\x75\x74\x61\x74\x69\x6f\x6e",
    "$\x76\x72",
    "\x6f\x74",
    "$\x32",
    "\x74\x70",
    ... ];

// Using the opaque strings in the source code
window[_[132]][_[65]]
```

Figure 3. This code fragment was taken from source code classified as Canvas fingerprinting by the dynamic analysis, but not the static analysis. This code is similar to much of the code in its classification group. Even the use of the underscore for the variable name is consistent across different scripts. These lists typically have lengths of a few hundred. The lists elements, once decoded, are often keywords which signify usage of the Canvas API, such as `fillText` and `toDataURL`.

A large number of scripts which were outside of this group which are also performing Canvas fingerprinting. Thus the intersection of these two analyses has a significant quantity of false negatives. These false negatives are largely due to code obfuscation and unexecuted code.

---

13Lines refers to the number of lines after the script is passed through prettier. Most scripts a web client receives will be minified to contain no newlines.
4. ANALYSIS METHOD AND RESULTS

Figure 4. This code fragment was taken from source code classified as Canvas fingerprinting by the dynamic analysis, but not the static analysis. This code is an example of a more advanced obfuscated technique, where string literals which are not intuitively decodable are defined, and then elsewhere in the code obscure arithmetic and/or bitwise operations are performed. As shown, such string literals are often a random assortment of ascii characters, including control characters. We manually escaped the control character in their code into the form \u{XX}, as they are not printable. Note the string literals are typically hundreds of lines, and arithmetic operations thousands of lines.
There were 699 site, script pairs which were classified as Canvas fingerprinting by the dynamic analysis, but not the lexical analysis. These 699 site, script pairs contained 486 unique scripts. As these scripts were not classified as fingerprinting by the lexical analysis, obfuscation of the method names is occurring. Many scripts share a structure similar to that shown in Figure 3. In this obfuscation strategy, an array of strings as escape characters is defined at the beginning of the script, then referenced later in the script. All other scripts contain a more advanced forms of obfuscated, in which we not able to decipher the functionality of a script, without the dynamic data associated with it. These more advanced forms of fingerprinting are not a monolith, but all follow a similar regime with strings literals of seemingly random characters and hard to follow arithmetic and bitwise operations. This obfuscation technique is similarly obscuring away the method names into string literals. However, it then using some opaque decoding mechanism (via the arithmetic and bitwise operations) to transform the seemingly random string literals into the desired method names. An example of script we found in this group following this obfuscated strategy is shown in Figure 4.

The presence of this obfuscation motivates a analysis which can overcome this opacification of functionality. To overcome the escaped characters, a lexical analysis may first unescape all escaped characters, then look for keywords of interest.

As the source code in this group is obfuscated to various degrees, the dynamic data recorded in the most useful in verifying a site, script pair was truthfully performing Canvas fingerprinting. Just 4 suspicious strings, passed as arguments to $\text{fillText}^{14}$ accounted for 683/699 of the elements in the group. Manually sifting through the dynamic data for the remaining 16 elements finds all of them to be fingerprinting, either by existence of other suspicious strings passed to $\text{fillText}$, ‘fp’ in the script name, and/or other forms of fingerprinting present in the dynamic data. Thus, all elements in this group were truthfully fingerprinting, with no false positives present (for the dynamic analysis). The degree to which four strings account for nearly all of the

\footnote{String which started with \texttt{Cwm fjordbank glyphs vext quiz}, \texttt{Hel$ \& \ 76\%}$ \{mZ+\# \%\}, \texttt{<\%nv45. F1n63r,Pr1n7inS!, or !H71JCaj)}}

\# 10\#.
Canvas fingerprinting occurring speaks to the widespread fingerprinting library usage. Additionally, though it was not necessary to manually confirm any of the site, script pairs were fingerprinting, looking at the lack of a Canvas element displayed upon site load can also be used to confirm a script is not legitimately using the Canvas API.

There were 797 site, script pairs which were classified as Canvas fingerprinting by the lexical analysis, but not the dynamic analysis, even though some instrumented JavaScript was recorded. These 797 site, script pairs contained 715 unique scripts. A brief look into a random sample of two dozen of the scripts finds only one third of them to be performing fingerprinting. A manual analysis of these scripts has been the most difficult group to definitively come to a conclusion on. Of the scripts manually analyzed, most of them contained calls to `fillText` in the source code which are never executed, thus not showing up in the dynamic data. Reading through the source code, it is often difficult tracking information flows of Canvas elements, and strings passed to `fillText`.

As with the previous groups, much of the scripts can be grouped into slight variations of the same library. FingerprintJS [13], ClientJS [38], and PerimeterX (now HUMAN) [19] were all found amongst this group. However, not all of these libraries seem to be fingerprinting. We have found libraries which abstract on top of the Canvas API, such as Chart.js [7], HighCharts [18] and EaselJS [17] several times in this group of scripts. The featureful use of the Canvas API presents a unacceptable level of false positives amongst this group.

There were 1,093 site, script pairs which were classified as Canvas fingerprinting by the lexical analysis, but no instrumented JavaScript was recorded. These 1,093 site, script pairs contained 932 unique scripts. These scripts lack of dynamic data implies either that the script did not execute, or that the script did execute, but did not touch any of the instrumented Web APIs. This group of site, script pairs are very similar to the group which were classified as Canvas fingerprinting by the lexical analysis, but not the dynamic analysis. By nature of these scripts being classified as Canvas fingerprinting by the lexical analysis, there is mostly likely code paths which contained part of the Canvas API which was not executed.
4. ANALYSIS METHOD AND RESULTS

<table>
<thead>
<tr>
<th>Form</th>
<th>Length of Hex Code</th>
<th>Unicode Range supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>\xXX</td>
<td>2</td>
<td>U+0000 - U+00FF</td>
</tr>
<tr>
<td>\uXXXXX</td>
<td>4</td>
<td>U+0000 - U+FFFF</td>
</tr>
<tr>
<td>\u{X} - \u{XXXXX}</td>
<td>16</td>
<td>U+0000 - U+10FFFF</td>
</tr>
</tbody>
</table>

Table 2. The three forms of an escaped character in JavaScript

Just like the other categories of scripts, most scripts found are a part a handful of libraries. A new library found very commonly in this group is Cloudflare’s JavaScript detections. [8] This library is part of a anti-bot service Cloudflare provides. They claim to use Google’s Picasso fingerprinting technique [6]. We were able to find scripts that was part of this library, as Cloudflare states, there JavaScript sources will be in paths starting with https://www.example.com/cdn-cgi/challenge-platform/... Every single script of the form:

https://www.example.com/cdn-cgi/challenge-platform/*/*/scripts/pica.js

was part of this group with no dynamic data, but classified as fingerprinting by the lexical analysis. This library accounted for 310 of the 1,093 (website,script URL) pairs in this group.

2.3. Results from unescaping escaped characters. Based on the prevalence of the obfuscation technique where string literals had their characters ‘escaped’ in the various forms shown in Table 2, we implemented a pre-processing step which would un-escape escaped characters before being searched by the lexical analyzer. We use the exact same lexical analysis as described earlier in this section. The process to unescape the escaped characters in a piece of JavaScript source code went as follows

15The *'s represent a singular character. We found the characters are typically h/b or h/g. We assume pica stands for picasso.

16The format was taken from the Mozilla MDN web documentation here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String#escape_sequences
4. ANALYSIS METHOD AND RESULTS

(1) Find an occurrence of a escaped character as a instance of one of the three rules shown in Table 2

(2) Replace the occurrence of the escaped character with the character which it’s hex code corresponds to

(3) Repeat for all escaped characters

This process was implemented using python’s \texttt{re.sub}, using a regular expression to search for occurrences of a escaped character, then specifying the replacement function to take a escaped character string, and return its associated character. Notably we are not parsing the script into a AST and only applying our replacements to string literals in the AST. As previously mentioned in our background previous static analyses, js-beautify, a code formatter, can also unescape strings. We opted against using it for this purpose, as we found the performance to be detrimental, taking about a minute per website, when tested on a small crawl\footnote{A minute per website would be 7 days on our crawl of 10,000 websites. Our result on a minute per website was a rough estimate from a crawl of three popular websites. However the order of magnitude would still scale the analysis from a task which took minutes to a task which took days, which would have been a major inconvenience.} We are confident, but cannot guarantee, that the only instances of a pattern which matches the form of a escaped character would be within a string literal. For our lexical analysis, whether our process has mapped the source code to functionality exactly identical code is irrelevant, as we only care to check the existence of keywords.

Table 3 presents a numerical summary of the results from the dynamic and lexical analysis. These results are best understood in comparison to our previous results without unescaping strings in Table 1. As the dynamic analysis did not change, the only changes are entries moving within the same column (i.e. changing their lexical analysis result) from no to yes. The change of unescaping character lexically classified 321, 5, and 19 new elements as Canvas fingerprinting, which were dynamically classified as yes, no and N/A respectively. Notable regarding these changes are that a large number
4. ANALYSIS METHOD AND RESULTS

<table>
<thead>
<tr>
<th>Dynamic Analysis</th>
<th>Yes</th>
<th>No</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical Analysis</td>
<td>Yes</td>
<td>851</td>
<td>802</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>378</td>
<td>64,048</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>74</td>
<td>6,762</td>
</tr>
</tbody>
</table>

Table 3. Results of the lexical analysis (with strings unescaped) and dynamic analysis on 7,875 functional websites for Canvas fingerprinting. Note this is over the domain of (website, JavaScript script URL). A result of Yes signifies the analysis classified the site, script pair as fingerprinting. A result of No signifies the analysis classified the site, script pair as not fingerprinting. A result of N/A signifies the analysis did not see any data from the script, site pair.

(321) of the lexical analyses false negatives were avoidable with a simple fix of unescaping strings. Additionally, 24, previously unseen fingerprinting elements were found, not previously found by either the dynamic analysis or lexical analysis.

Beyond these 24 newly found site, script pairs, all of the scripts have been manually analyzed in prior portions of this section. From our discussion on the scripts which were classified as canvas fingerprinting dynamically, but not lexically, (without unescaping characters) we observed widespread escaped characters. The observations from this group was the motivation towards re-running the lexical analysis with characters escaped. The concrete numerical result that 321/699 site, script pairs in this group were escaping their method names vindicates the importance of unescaping strings in a lexical analysis.

All 24 new site, script pairs which were found by the lexical analysis, yet not found by the the dynamic analysis were manually confirmed to be fingerprinting. They were three distinctly different libraries/obfuscation style used among this group, pointing to
many actors independently choosing to escape characters as a obfuscation technique. We previously found no false positives among the elements classified as Canvas fingerprinting via our dynamic analysis. The 24 newly found elements not classified as Canvas fingerprinting via our dynamic analysis were also manually confirmed to be performing Canvas fingerprinting. Thus every script which had its lexical analysis results change to a yes once characters were unescaped was truthfully performing Canvas fingerprinting.

3. Canvas Font Fingerprinting

3.1. Characteristics of canvas font fingerprinting in static and dynamic crawl data. We used the criteria for Canvas font fingerprinting as specified in [11]. Their requirements were as follows

- The `CanvasRenderingContext2D.font` property is set to at least 50 distinct valid fonts.
- The `CanvasRenderingContext2D.measureText()` method is called at least 50 times on the same string.

The motivation behind these requirements are fairly straightforward. In order to use the Canvas API to detect fonts on the system, the `font` property need to be set to many fonts, following subsequent calls `measureText()` with a consistent string in order to detect font differences. The threshold of 50 was a heuristic used by the authors. We kept the threshold for 50 as we found that most scripts were either using a few to no fonts, or more than 50 fonts.

Our resulting dynamic analysis are the following conditions:

1. `CanvasRenderingContext2D.measureText` is called a minimum of 50 times on the same string
2. `CanvasRenderingContext2D.font` is set to a minimum of 50 unique values

Our resulting lexical analysis are the following conditions:

1. There exists the keyword `.measureText` in the source code
2. There exists the keyword `.font` in the source code
3.2. Results. Table 4 presents a numerical summary of the results from the dynamic and lexical analysis. In contrast to Canvas fingerprinting, Canvas font fingerprinting has much less prevalence, based off the dynamic analysis. This trend is in agreement from the findings of [11].

In performing a manual analysis to verify results and resolve discrepancies, we must more narrowly define what we consider Canvas font fingerprinting. The repeated extraction of the `TextMetrics`\(^{18}\) from the same piece text with a substantial\(^{19}\) number of fonts with the same font size and different font families, using alternating calls to `CanvasRenderingContext2D.font` and `CanvasRenderingContext2D.measureText()`. This is largely a slight alteration to the definition we used above.

\(^{18}\)The `TextMetrics` are what is returned by `measureText`.

\(^{19}\)Substantial is not a precise definition. However no script’s we found were on the fence between a number of fonts which no websites would legitimately simultaneously use, and legitimate use. Including signs of other forms of fingerprinting, concluding on a manual analysis was never indecisive.
There were 23 site, script pairs which were classified as Canvas font fingerprinting by both the dynamic and lexical analysis. These 23 site, script pairs contained 21 unique scripts. The dynamic requirements for Canvas font fingerprinting exist such that, outside of purposely creating a ‘mock’ false positive, a false positive is near unimaginable. As such, we confirmed all scripts to be performing Canvas font fingerprinting through manually analyzing the static and dynamic data. We observed noteworthy characteristics about the scripts in this group. We found comparatively, a greater number of boutique scripts which were dissimilar to each other. 5 of the 21 scripts seemed unique, not easily attributed to slight alterations of common previously seen scripts. The remaining 16 scripts are all slight variation of each other, with their obfuscation transpilers using different random sequences of letters and numbers for identifier names between the scripts. Additionally, they all define the same 1,071 fonts in arrays of string literals.

There were 66 site, script pairs which were classified as Canvas font fingerprinting by the dynamic analysis, but not the lexical analysis. These 66 site, script pairs contained 73 unique scripts. 65 of the 66 script, site pairs used at least one of two suspicious strings which cements their illegitimate use of the Canvas API. Beyond fulfilling the stringent dynamic requirements, all of the fonts assigned are of the same font size and different font families, as is expected for Canvas font fingerprinting. All of the scripts are within the category of advanced obfuscation techniques. Even the use of the Canvas API was unable to be detected from a human reading the source code (including unescaping escaped string literals when relevant) in all but 5 of the 73 scripts. Being aware of the use of the string "Hel$&?6%){mZ+#@", 26 of the 73 scripts included this string within a string literal in the source code. Without knowledge about the common usage of this

---

20 They actually all defining 5 arrays of string literals, containing a total of 1,071 fonts, but this is pedantic.

21 This is contrary to most other situations, where duplicate scripts cause less distinct JavaScript source code than site, script pairs. However, one site, script pair may involve multiple requests with different parameters, thus returning different scripts.

22 Strings starting with "Cwm fjordbank glyphs vext quiz" or "Hel$&?6%){mZ+#@"
4. ANALYSIS METHOD AND RESULTS

"MrAgg_ZStcKalIOa51EDExdZsFNd",
"documentMode",
"He1$&?6%}{mZ+#@\uD83D\uDC7A",
"8gDnYGHbNTFBBgg",

Figure 5. Code fragment taken from a script classified as Canvas font fingerprinting dynamically and not lexically

string in these fingerprinting script (presumably because they are from the same library, open source or proprietary) this string looks like one of many random string literals as shown in Figure 5.

Thus all\textsuperscript{23} of the scripts classified as Canvas font fingerprinting by the dynamic analysis were true positives.

There were 1,243 site, script pairs which were classified as Canvas font fingerprinting by the lexical analysis, but not the dynamic analysis, even though some instrumented JavaScript was recorded. These 1,243 site, script pairs contained 782 unique scripts. Notably of the 1,243 site, script pairs, only 509 of the pairs have any runtime execution of portions of the Canvas API, and 121 of those call both `CanvasRenderingContext2D.font` and `CanvasRenderingContext2D.measureText`. Thus the majority of this group of scripts involve code paths which call those two methods, yet did not execute at runtime upon site load. In the group of 121 site, script pairs using `CanvasRenderingContext2D.font` and `CanvasRenderingContext2D.measureText`, most of the usage seemed to be toward legitimate site functionality. The strings passed which had their text metrics measured were a variety of human comprehensible strings with a reasonable connection to strings which might be displayed on a given site. In contrast, Canvas font fingerprinting scripts are querying the text metrics of the same string, which is often not human comprehensible (a seemingly random sequence of characters).

\textsuperscript{23}Excluding the inline ones without static data saved.
There were however, another category in there 121 scripts dynamic data, whose made many repeated calls to to `.measureText` on the same suspicious string. However, unlike as is required for Canvas font fingerprinting, it was not alternated with changes to the font. Either a dynamic instrumentation failure or poorly designed JavaScript are equally plausible to blame. The majority of site, script pairs did not make calls to CanvasRenderingContext2D.font and CanvasRenderingContext2D.measureText at run-time, yet contained .font and .measureText in their source code, representing a untraveled code path. Looking at the source code, a large amount of the usage seems to be legitimate, often for the purpose of checking if one or two fonts exist before using them. Looking at the source code of 200 random scripts, only 35 of them were truthfully performing Canvas font fingerprinting. All 35 scripts perform Canvas font fingerprinting were exactly the same as shown in Figure 6.

There were 702 site, script pairs which were classified as Canvas font fingerprinting by the lexical analysis, but no instrumented JavaScript was recorded. These 702 site, script pairs contained 572 unique scripts. Performing a manual analysis on a random sample of 200 scripts in this group, 11/200 were truthfully performing Canvas font fingerprinting. A tell of a font fingerprinting script is a large list of different fonts. For every script we found with a large list of fonts, we were able to manually verify that they were in fact font fingerprinting. This manual verification, as previously done relied, on observing occurrences of other forms of fingerprinting, observing non human readable text being measured, and observing the text metrics were used only for a comparison to other text metrics.

4. WebGL Parameter Fingerprinting

4.1. Characteristics of WebGL parameter fingerprinting in static and dynamic crawl data. Our requirements for performing WebGL parameter fingerprinting was querying both the UNMASKED_VENDOR_WEBGL and UNMASKED_RENDERER_WEBGL parameters. This is performed as follows:
4. ANALYSIS METHOD AND RESULTS

Kj = "monospace; sans-serif; serif; Andale Mono; ... ".split(
    "",
  ),

rq = t(function (a) {
  a = jb(a)("canvas");
  var c = n(a, "getContext");
  if (!c) return null;
  try {
    var b = L(c, a)("2d");
    b.font = "72px mmmmmmmmmlli";
    var d = b.measureText("mmmmmmmmmmlli").width;
    return function (e) {
      b.font = "72px " + e;
      return b.measureText("mmmmmmmmmmlli").width === d;
    };
  } catch (e) {
    return null;
  }
}),

Figure 6. A piece of source code, classified as Canvas font fingerprinting lexically but not dynamically. This code fragment was common. The string of fonts terminated by ... continues for about 900 characters. This script appear to (at the very least) be trying to perform Canvas font fingerprinting.

- a WebGLRenderingContext makes a call to getParameter with the GLenum value of 0x9245, accessing the UNMASKED_VENDOR_WEBGL constant.
4. ANALYSIS METHOD AND RESULTS

- a WebGLRenderingContext makes a call to getParameter with the GLenum value of 0x9246, accessing the UNMASKED_RENDERER_WEBGL constant.

We will consider a script to be performing WebGL parameter fingerprinting if either of the two actions listed above are performed.

Translating this into a dynamic analysis went as follows. We dynamically instrumented the WebGLRenderingContext object. We look through the tabular dynamic data for the calls to getParameter with the function parameters of the two enum values corresponding with our parameters of interest. If either one of the two parameters (UNMASKED_VENDOR_WEBGL or UNMASKED_RENDERER_WEBGL) Our resulting dynamic analysis are the following conditions:

(1) WebGLRenderingContext.getParameter is called with 37445 or 37446 as its singular argument

For a lexical analysis, we got more creative than searching for the method name used dynamically. We opted to consider any instance of the three phrases, UNMASKED_VENDOR_WEBGL, UNMASKED_RENDERER_WEBGL, or WEBGL_debug_renderer_info, as evidence of WebGL parameter fingerprinting. These three phrases, all containing ‘WebGL’ within them, would be highly unlikely to be used outside of accessing either of the parameters of interest. Though there are known GLenum values which can be passed to getParameter, the activation of the WEBGL_debug_renderer_info with a call to .getExtension() is required. Additionally, in previously seen fingerprinting code which was performing Canvas fingerprinting or Canvas font fingerprinting, these three keywords were often present in a part of the code whose sole functionality was obtaining these two WebGL parameters of interest. Our resulting lexical analysis are the following conditions:

(1) There exists the keyword WEBGL_debug_renderer_info, UNMASKED_VENDOR_WEBGL, or UNMASKED_RENDERER_WEBGL in the source code

\footnote{We did not however, instrument the WebGL2RenderingContext object. We are unsure as to the degree of the impact doing so would have caused.}
4. ANALYSIS METHOD AND RESULTS

### Dynamic Analysis

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2,277</td>
<td>183</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>63,762</td>
<td>198,140</td>
</tr>
</tbody>
</table>

| N/A | 10 | 6,826 |

**Table 5.** Results of the lexical analysis and dynamic analysis on 7,875 functional websites for WebGL parameter fingerprinting. Note this is over the domain of (website, JavaScript script URL). A result of Yes signifies the analysis classified the site, script pair as Fingerprinting. A result of No signifies the analysis classified the site, script pair as not fingerprinting. A result of N/A signifies the analysis did not see any data from the script, site pair.

#### 4.2. Results

The results of the dynamic and lexical analysis of WebGL parameter fingerprinting are listed in Table 5.

There were 0 site, script pairs which were classified as WebGL parameter fingerprinting by both the dynamic and lexical analysis. This results from all scripts querying the parameters of interest being obscenely obfuscated, and all scripts containing the keywords associated with the parameters of interest not executing the retrieval of such parameters at run time. Without thorough testing of a websites entire runtime behavior\(^{25}\), we cannot conclude on why the code paths which would have queried these parameters did not execute.

There were 40 site, script pairs which were classified as WebGL parameter fingerprinting by the dynamic analysis, but not the lexical analysis. These 40 site, script pairs which were classified as WebGL parameter fingerprinting by the dynamic analysis, but not the lexical analysis. These 40 site, script pairs which were classified as WebGL parameter fingerprinting by the dynamic analysis, but not the lexical analysis. These 40 site, script pairs which were classified as WebGL parameter fingerprinting by the dynamic analysis, but not the lexical analysis. These 40 site, script pairs which were classified as WebGL parameter fingerprinting by the dynamic analysis, but not the lexical analysis.

\(^{25}\)This would require a depth of data well beyond the history of API calls which we are using as dynamic data. As a proposed idea, one could use a JavaScript debugger with breakpoints to manually determine why a script is deciding not to execute a specific a code path.
4. ANALYSIS METHOD AND RESULTS

script pairs contained 46 unique scripts. All of the 40 script, site pairs were solely using the WebGL API to gather the two parameters of interest. Additionally 37 of these 40 scripts are passing "Cwm fjordbank glyphs vext quiz" to CanvasRenderingContext2D.fillText(), among other fingerprinting activities. Of these 37 scripts, there source code all contained the the exact same string list with 398 elements, save for one random sting and one value which was either "Legacy" or "Lax". This indicates that perhaps before the obfuscation all of these scripts were slight version changes of each other. With shared references to Selenium, search engine, web crawlers, and "automationCheck", we believe these scripts to be part of some form of bot detection / mitigation.

The other three script, site pairs are likely to be fingerprinting due to their navigator property’s and media queries fingerprinting performed. These scripts did have comprehensible words within its source code in the form of a list of string literals. However, references to anything related to WebGL API or Canvas API are non existent within the source code. Thus the use of these libraries (and perhaps others) was obfuscated away. The lack of anything in the source code with a resemblance to the WebGL library explains the failure of the lexical analysis to successfully classify these scripts as WebGL parameter fingerprinting. Additionally however, it leaves no immediate insight into improvements to a future lexical or static analysis, as the obfuscation makes it hard to envision a methodology which would, (at the bare minimum) detect the use of the WebGL from this static data.

In contrast to other fingerprinting types, the lexical analysis was significantly more likely to classify a script as performing WebGL parameter fingerprinting if it did have dynamic data, rather than if it did not have dynamic data. The question remains, if parts of the script did executed interactions with parts of the Web API that we instrument, why did none of the 2,277 scripts lexically classified not execute the querying the parameters of interest (UNMASKED_VENDOR_WEBGL and UNMASKED_RENDERER_WEBGL).

There were 2,227 site, script pairs which were classified as WebGL parameter fingerprinting by the lexical analysis, but not the dynamic analysis, even though some
instrumented JavaScript was recorded. These 2,227 site, script pairs contained 1,365 unique scripts. By definition of this group a script were trying to extract the UNMASKED_VENDOR_WEBGL or UNMASKED_RENDERER_WEBGL properties, on the basis of these keywords existence in the scripts, or the WebGL extension WEBGL_debug_renderer_info. Thus there likely exists a code path which extracts at least one of the two WebGL parameters which was not executed during runtime. In fact, only 2 script, site pairs of the 2,227 were instrumented using the WebGL API at runtime. Further looking into the dynamic data finds 177 site script pairs passing ‘Cwm fjordbank glyphs vext quiz’ to CanvasRenderingContext2D.fillText(), and 294 site, script pairs were classified as Canvas fingerprinting dynamically. These signs of other forms a fingerprinting give a gauge as how many of the 2,227 script, site pairs are fingerprinting scripts, as the existence of other forms of fingerprinting is typically a good indication of any given type of fingerprinting. Additionally, the further begs the question as to why these scripts Canvas fingerprinting functionality executed at runtime, but the WebGL parameter fingerprinting did not. Manually analyzing the source code from this ground finds 106 of the scripts fingerprinting, out of a random sample of 200 scripts. In our source code analysis, every script was querying the renderer and vendor properties. Our keywords that we search for were never found being used for any other purpose. Typically all three of the keywords would be present, occasionally one or two of the keywords would be obfuscated away. Every script was extracting

26These two scripts, which were from the same site, seemed to have some sort of technical difficulty in the instrumentation of the WebGL API. Calls to WebGLRenderingContext.getParameter were made (as well as WebGLRenderingContext.bufferData and WebGLRenderingContext.readPixels), however, they were passed no arguments.

27As previously discussed, any human querying the WebGL renderer and vendor would use all three keywords. However, WEBGL_debug_renderer_info can be easily obfuscated away, as it is string to be passed to WebGLRenderingContext.getExtension. UNMASKED_VENDOR_WEBGL and UNMASKED_RENDERER_WEBGL can be obfuscated away from the source code by replacing them with their enum values when calling WebGLRenderingContext.getParameter. Their enum values are 37445 and 37446 respectively.
these parameters, yet we need to judge whether the extraction of those parameters were for the purpose of fingerprinting or a legitimate purpose. The characteristics we use to make this decision were other forms of fingerprinting in the script, and extensive WebGL usage. Extensive WebGL usage implies it is highly likely the script is using WebGL to perform some legitimate functionality. In the 200 scripts we looked at, a few dozen were using WebGL vendor and renderer as well as numerous additional WebGL parameters as part of the browser fingerprint. This presents the question as to if other WebGL parameters provide entropy on a web user.

There were 183 site, script pairs which were classified as WebGL parameter fingerprinting by the lexical analysis, but no instrumented JavaScript was recorded. These 183 site, script pairs contained 150 unique scripts. A manual analysis of these 150 scripts reveals 115 of them to be fingerprinting. Additionally, in this group, a lot of code was packed into a string literals, which presumably get evaluated later. This makes a manual analysis more difficult in our workflow, as our code formatter cannot format the JavaScript source code which is in a string literal. The lexical analysis was able to work just the same, as the keywords are still present in the string literals.
CHAPTER 5

Conclusion

In this thesis we set out to compare different methods of analyzing the presence of browser fingerprinting on a website. We implement a dynamic and lexical analysis, and subsequently deploy them on a large scale web crawl. We give in depth detail to describe the fingerprinting scripts actively deployed to the web, as well as characteristics of these scripts which allow for their detection or evasion from our analyses. We largely find a lexical analysis fail to detect a fingerprinting script in the presence of code obfuscation, and a dynamic analysis fail to detect a fingerprinting script in the presence of unexecuted code. Additionally, the lexical analysis suffers from a great deal of false positives, as it does not differentiate legitimate and fingerprinting uses of a subset of the Web API. We observed the prevalence of escaping the characters in a string literal as a relatively simple form of obfuscation, and modified our lexical analysis to unescape strings as a preprocessing step. As we expected from the observation of the widespread presence escaping characters in string literals, this led to a substantial improvement in the true positives of the lexical analysis.
Bibliography


[38] J. Spirou. ClientJS. [https://github.com/jackspirou/clientjs/tree/v0.2.1](https://github.com/jackspirou/clientjs/tree/v0.2.1), 2021.


2018.


APPENDIX A

js_instrument_settings

The settings passed to OpenWPM specifying which APIs to dynamically instrument.

In OpenWPM, these are passed as the js_instrument_settings attribute to a browser parameters object. ¹

```javascript
[
  "RTCPeerConnection",
  {
    "HTMLCanvasElement": {
      "propertiesToInstrument": [
        "getContext",
        "height",
        "width",
        "toDataURL"
      ]
    }
  },
  "CanvasRenderingContext2D",
  {
    "window": {
      "propertiesToInstrument": [
        "matchMedia"
      ]
    }
  }
]
```

¹The documentation for OpenWPM’s JavaScript instrumentation may be read here: https://github.com/openwpm/OpenWPM/blob/master/docs/Configuration.md/#js_instrument
{ "window.performance": { "propertiesToInstrument": [ "memory" ] }, "window.screen": { "recursive": true, "depth": 1 } }, { "window.navigator": { "excludedProperties": [ "mimeTypes","plugins","userAgent" ], "recursive": true, "depth": 1 } }, "WebGLRenderingContext" ]