Bachelor thesis
Computing Science

Radboud University

Towards finding browser fingerprinters through automated static analysis of JavaScript code

Author:
Bart van Vulpen
s4800346

First supervisor/assessor:
Dr. Ir. Hugo Jonker
`hugo.jonker@ou.nl`

Second assessor:
Dr. Ir. Erik Poll
`erikpoll@cs.ru.nl`

June 14, 2020

Abstract

Online tracking has become a big part of online life, with some methods being more insidious than others. In this thesis we take a look at the methods currently being used by browser fingerprinters, we do this by manually analysing fingerprinting JavaScript. We then attempt to separate browser fingerprinting JavaScript from non-browser fingerprinting JavaScript by statically analysing the JavaScript. This results in a separation of fingerprint-like and non-fingerprint-like JavaScript. We then manually analyse the fingerprint-like scripts further to find 13 previously unknown commercial browser fingerprinters.

Contents

Chapter 1

Introduction

Online tracking has become a big part and problem of online life. Particularly insidious is browser fingerprinting, as this is done server side and the user does not notice this unless they are looking for it. This can be first-party tracking, which can be done for good reasons as re-authentication and fraud prevention. First-party tracking does not have a big impact on privacy while it does improve the functionality of the website for the individual user, because of this it is often not seen as a bad thing. That is as long as the first parties do not share the gathered information with third parties.

However, some of these companies also track people on websites that they do not own, known as third-party tracking. Browser fingerprinting is one of the ways this is done. This is done by collecting information about a device through the browser. When a user loads a page this information can be collected and used to construct a fingerprint. Then when the user loads another page on the same device it will give the same fingerprint and this way the device can be tracked online. JavaScript is often used for this purpose.

There are tools that attempt, and (partially) succeed, to block some form of third-party tracking, like FP-Block [TJM15] and PriVaricator [NJL15] which both attempt to block browser fingerprinting. But to be able to block third-party tracking it first has to be known what methods of tracking are being used by the companies that do this. For which we need to find these companies that are tracking. Which can be quite some work if done manually.

This gives us the following research questions:

- Which methods are used by fingerprinters at the moment?
- Can we find these fingerprinters in an automated way?

There are overviews of companies that do some kind of third-party tracking and what attributes they use, but most of these are about a single type of fingerprinting, like the one in Tracking the pixels [FBLS18]. There are also overviews which show which companies track and what they track, like the one in FP-Block [TJM15], but these are not up-to-date as these companies change, like BlueCava[1] which merged with a different company, or AddThis[2], which was bought by a different company[3]. This may change the methods they use to fingerprint, and thus is an up-to-date overview needed to know what methods are used and to effectively be able to block third-party fingerprinting.

We make the following contributions:

- An overview of the attributes used by commercial fingerprinters, as previously given in table 1 of Fp-block [TJM15].
- A systematic way to differentiate between fingerprinting and non-fingerprinting JavaScript based on our findings from manual analysis.
- We introduce a method using static JavaScript analysis to find previously unknown commercial browser fingerprinters.

In chapter 2 we will discuss the background and related work of this thesis. Chapter 3 contains the methodology we used. Chapter 4 goes into how manual analysis of potential fingerprinters works. Chapter 5 contains the design of our automated tool to analyse potential fingerprinters. In chapter 6 we use this tool to find fingerprinters. Chapter 7 contains our conclusions.

[1]https://bluecava.com/
[2]https://www.addthis.com/
[3]https://techcrunch.com/2016/01/05/oracle-addthis/

Chapter 2

Background & Related work

2.1 Background

Browser Fingerprinting is used to identify an individual user through properties and settings of the devices they use. When talking about browser fingerprinting we talk about this fingerprinting happening through a browser. The browser already provides a server with some information in the HTTP header. And with script execution in the browser a lot more information can be gathered, also information not related to the browser like screen resolution and OS. With all this information a (unique) identifier, a fingerprint, can be created for a user. This fingerprint can then be used to identify the user on different sites and even different browsers. Unlike cookies, which are stateful, fingerprints are stateless. This means that cookies can be easily read and deleted by a user, as they are stored on the client side. Fingerprints are not stored on the client side, which means a user can not simply read or delete fingerprints.

A *web scraper* is a program that automatically visits sites and collects a large amount of data from the visit, like cookies, HTTP requests and responses, and JavaScript that gets run during the visit.

A *web crawler* is a program that automatically visits sites and collects a copy of the visited sites for later processing.

Commercial fingerprinting is offering a fingerprinting service, for commercial purpose. This can be done for advertising purposes but also for fraud prevention or authentication.

The *fingerprinting surface* is the collection of all methods that can be used to fingerprint devices. This surface can not be made complete as a small change can add or delete a certain method.

2.1.1 Glossary

A *pay-level domain* is the domain name that a consumer or business can directly register, and which consists of a subdomain of a public suffix or effective top-level domain(e.g. *.nl* or *.co.uk*). It is also called the 2nd Level Domain, Complete Domain or Host Domain.

A *backlink* for a given web resource is a link from a different website which refers to that web resource.

A *class c subnet* is a subnet with the three most significant bits set to 1, 1, 0 and has the next 21 bits designated to number the networks. Also known as the IPv4/24 addresses.

An *abstract syntax tree* is a tree representation of the abstract syntactic structure of code, where each node of the tree denotes a construct occurring in the code.

2.2 Related work

2.2.1 Web Measurement Platforms

There are different platforms available which can automatically measure some form of privacy, and collect data. Here we name a few of the available platforms which we could have used to try and find fingerprinters.

commoncrawl[1] uses an Apache Nutch based crawler. It also has a publicly available webcrawl dataset online. However, this crawler does not execute JavaScript or other dynamic content.

webXray [Lib15] is a PhantomJS based tool that measures HTTP traffic. Since it uses the stripped-down browser PhantomJS it has the potential to miss a large number of resource loads. Since it measures the HTTP traffic we could derive most JavaScript from this measurement and thus would be able to use this for our research.

[1]https://commoncrawl.org

FourthParty [MM12] is a Firefox plugin for instrumenting the browser. It does not handle automation. It only supports a subset of OpenWPM's instrumentation. We could use this to detect fingerprinting.

Chameloen crawler[2] is a Chromium based crawler. It utilizes the Chameleon browser Extension[3] to detect browser fingerprinting. It is automated, but only detects a small part of the fingerprinting surface.

Trackingobserver [Roe14] is a Chrome extension that detects, measures and block third-party web trackers. It does not use blacklists, but rather detects in-browser behaviour.

FPDetective [AJN$^+$13] is a framework for detection and analysis of browser fingerprinting. It focuses on detecting fingerprinting instead of relying on known fingerprinters. It only supports stateless measurements.

OpenWPM [EN16] is a web-scraper often used in research. It uses a full-fledged consumer browser, and selenium[4] to drive this browser. It is made to detect and characterize online tracking behaviour. We use OpenWPM later in this thesis for our research.

2.2.2 Top Site Rankings

Cisco Umbrella[5] publishes a list that gets updated daily[6]. This list contains one million entries, this lists can include any domain name. The domains get ranked on the traffic count of itself aggregated with the traffic count of all its subdomains. The traffic considered is DNS traffic to two DNS resolvers owned by Cisco Umbrella. They claim these DNS resolvers amount to over 100 billion daily requests from 65 million unique users[7]. The domains are ranked on the number of unique IP's that issue DNS queries for them. The DNS data collected is sampled and to reduce biases they apply 'data normalization methodologies', which takes in account the distribution of client IP's[8]. This method means that non-browser based traffic is also accounted for, which also means invalid domains are also included.

[2]https://github.com/ghostwords/chameleon-crawler

[3]https://github.com/ghostwords/chameleon

[4]https://www.selenium.dev/

[5]https://umbrella.cisco.com/

[6]https://s3-us-west-1.amazonaws.com/umbrella-static/index.html

[7]https://umbrella.cisco.com/blog/2016/12/14/cisco-umbrella-1-million/

[8]https://medium.com/cisco-shifted/cisco-umbrella-releases-free-top-1-million-sites-list-8497fba58efe

Majestic[9] publishes the 'Majestic Million' list[10], which gets updated daily. This list is mostly comprised of pay-level domains, but for some very popular sites subdomains are also included. The ranks are calculated based on backlinks to websites. These backlinks are obtained by a crawl of around 450 billion URLs over 120 days[11]. The ranking of sites is based on the number of class C subnets that refer at least once to the site[12]. This method of data collection means that only domains linked from other websites are considered, which implies a bias towards browser based traffic. However they do not actually count the amount of actual page visits. This means that the completeness of the data is affected by how their crawler discovers websites.

Quantcast[13] publishes a list of the most visited websites in the United States[14] daily. This list varies in size and before November 14 2018, it was usually around 520.000 mostly pay-level domains. This included sites with direct measurements of Quantcast trough a tracking script as well as sites where the traffic was estimated. However they stopped doing the estimations which reduced the size of the list to around 40.000 domains.

Alexa[15] publishes a daily updated list of the top 1 million websites. This was widely used in research. However, it was shown that this list is highly variable and not a reliable input for studying the web. The methods used by Alexa lead to a focus on sites that are visited in the top-level browsing context of a web browser. They also indicate that ranks worse than 100.000 are not meaningful in a statistical way. It also means that a small change in measured traffic can result in a large rank change[16], which negatively impacts the stability of the list.

Tranco [PvGT+19] is a research oriented top site ranking. This list is comprised of the other lists named here. The stability of this list is improved upon by averaging the ranks of domains on the lists over 30 days. It is also hardened against manipulation because of it being comprised of the four other lists over 30 days, which makes it harder to manipulate the rank of a site. The repeatability of studies is improved by archiving the daily lists and giving a permanent link to list of any given day. We use Tranco later in this thesis for our research.

[9]https://majestic.com/
[10]https://majestic.com/reports/majestic-million
[11]https://blog.majestic.com/company/majestic-launch-a-bigger-fresh-index/
[12]https://blog.majestic.com/development/majestic-million-csv-daily/
[13]https://www.quantcast.com/
[14]https://web.archive.org/web/20070705200342/http://www.quantcast.com/
[15]https://www.alexa.com/
[16]https://support.alexa.com/hc/en-us/articles/200449744

Chapter 3

Methodology

In this chapter we talk about the different methods available to 1. collect the JavaScript files we need, which is in section 3.1, 2. find potential fingerprinters, which is in section 3.2, and 3. determine which of the potential fingerprinters actually fingerprint, which is in section 3.3.

3.1 Data Collection

3.1.1 Why collect data?

To be able to find new fingerprinters we need to know what methods are being used. To see what methods fingerprinters are using at the moment we need recent data, as the methods used might have changed from previous findings. There is recent data online of web-crawls like the one by Common Crawl[1], but these do not include the JavaScript we need. Thus to be able to find new fingerprinters based on the methods used at this moment we need to collect new data.

3.1.2 The data to collect

To collect data we need a list of websites. To make the data related to what a normal user might experience we use a top ranking list of websites. There are multiple lists available that rank websites on popularity based on their own methods. Following are a few options.

[1]https://registry.opendata.aws/commoncrawl/

Alexa[2] publishes a daily updated list of the top 1 million websites. This "Alexa Top 1 Million" was widely used in research. However, it was shown by Le Pochat et al. [PvGT$^+$19] that this list is highly variable and not a reliable input for studying the web. The panel on which Alexa bases their data after "data normalization"[3], is claimed to consist of millions of users. However, Le Pochat et al. [PvGT$^+$19] suggest that there are at most 1 million users. The methods used by Alexa lead to a focus on sites that are visited in the top-level browsing context of a web browser. They also indicate that ranks worse than 100.000 are not meaningful in a statistical way. It also means that a small change in measured traffic can result in a large rank change[4], which negatively impacts the stability of the list.

Tranco[5] is a research oriented top site ranking made by Le Pochat et al. [PvGT$^+$19]. This list is comprised of the lists named here and the lists named in section 2.2.2. The stability of this list is improved upon by averaging the ranks of domains on the lists over 30 days. The score for each domain are proportionally rescaled to the same range, to account for the difference in length of the lists and the fact that some include subdomains and others do not. It is also hardened against manipulation because of it being comprised of the four other lists over 30 days, which means that to get the same manipulation in the Tranco list you would either have to influence multiple lists, get a lot higher on one list, or hold the manipulation longer. The repeatability of studies is improved by archiving the daily lists and giving a permanent link to list of any given day.

For this thesis we chose to use the Tranco list over any of the other lists because it gives a more representative of what a normal user might encounter, improved repeatability of the research, and is specifically designed to avoid the pitfalls of the other lists.

3.1.3 Data collection tool

To automate the collection of data we make use of a web-scraper. There are different web-scrapers available that we could use for this thesis. Following are a few good options that are available.

FPDetective[6] is a framework for detection and analysis of browser fingerprinters made by Acar et al. [AJN$^+$13]. It focuses on detecting browser

[2]https://www.alexa.com/
[3]https://support.alexa.com/hc/en-us/articles/200449744
[4]https://support.alexa.com/hc/en-us/articles/200449744
[5]https://tranco-list.eu/
[6]https://github.com/fpdetective/fpdetective/

fingerprinting itself instead of relying on information about known fingerprinters or third-party lists of fingerprinters. It is build for the analysis of browser fingerprinters and much of the instrumentation is built to support that. It uses two different instrumented browsers, namely PhantomJS[7] and Chromium[8]. FPDetective only supports stateless measurements.

OpenWPM[9] is an open-source web privacy measurement tool made by Englehardt et al. [EN16]. It uses an automated version of a full-fledged consumer browser, Firefox[10]. It also support automatic recovery from failures of this browser. It uses Selenium[11], which is a cross-platform web driver for Firefox, Chrome, Internet Explorer and PhantomJS. This also give the possibility to use different browsers with OpenWPM. Because a consumer browser is used, all the technologies of a typical user are supported by the measurements. PhantomJS does not have support for some of these technologies, like WebGL and browser-plugins, so by using this browser you miss part of the technologies in your measurements. OpenWPM stores compressed web content in a LevelDB, this includes all executed JavaScript. Since all executed JavaScript is stored there is no need to do extra things to avoid losing JavaScript due to in-line JavaScript and other tricks.

For this thesis we chose to use OpenWPM as it collects all executed JavaScript for us, giving us no worries about in-line JavaScript and other tricks fingerprinters might to do hide their code. It also gives gives us all the fingerprinters a normal user might encounter since it uses a full fledged browser, not missing out on certain technologies as a result of a stripped-down browser. And since it uses selenium this gives a more generic approach than the other scrapers.

3.2 Finding potential fingerprinters

To find unknown fingerprinters we first need to know how to recognise fingerprinters. To be able to recognise fingerprinters we need to know what methods we can recognise them by. For this we need to have JavaScript code of the fingerprinters. After collecting this code we can use the known fingerprinters to see how well our tool finds known fingerprinters. We can also see if our tool misses a lot of known fingerprinters or not. Thus we can make an estimate on how well our tool works.

[7]https://phantomjs.org/

[8]https://www.chromium.org/

[9]https://github.com/mozilla/OpenWPM

[10]https://www.mozilla.org/firefox/new/

[11]https://www.selenium.dev/

We are looking for commercial fingerprinters. Since they are commercial they will most likely advertise their services online. We can then find these advertisements by googling for them. Since we know from previous papers that fingerprinting is being used for fraud detection and prevention services [Lap17], and for authentication purposes [AvO16] we can use Google to find providers of these services, which resembles a method used by Jonker et al. [JKV19]. All the providers we find this way are potential fingerprinters, but not all will actually be fingerprinters. We can also not always check this by looking at what they state on their websites as some fingerprinters will avoid saying they are fingerprinting as it is often looked bad upon. We can also find more potential fingerprinters by googling for things like 'personalized ad services online' as these kind of services also use fingerprinting [NKJ$^+$13]. Again we do still need to confirm if they are actually fingerprinting or not.

Another way to find potential fingerprinters is to look at the fingerprinters previous papers, like FP-Block [TJM15] and PriVaricator [NJL15], have found. This will give us companies that are known to have fingerprinted in the past. Some of these papers also provide the fingerprinting code they have found online of some of these companies. We can then check if this code is still available as an indicator if these companies are still fingerprinting or not. But we also have to actually check if these companies still fingerprint, as they might have changed the code, but still fingerprint, or it might be that the code is still available but not used anymore.

We can also find more potential fingerprinters while collecting the code of the potential fingerprinters found with the previously named methods. While collecting the code of potential fingerprinters we scrape the web to sites that are using the services of the potential fingerprinter. These sites can often be found online by googling which sites use a certain service of the potential fingerprinter. While looking at the JavaScript collected from these sites to see if we have found some scripts of the potential fingerprinter we might find some other scripts with terms like 'ad' or 'tracking' in their name. This makes these scripts look suspicious and makes them potential fingerprinters.

For this thesis we mostly rely on the first method of simply using Google to find companies advertising their services. This choice was made because this method provides us with companies that are suspected to fingerprint at this moment, instead of companies that have fingerprinted, but might not do it anymore. The first method is also a lot more directed than the last method in the case of scraping random sites. We did however also consider and looked at the previously named fingerprinters by other papers.

3.3 Determining who is a fingerprinter and who is not

3.3.1 Preparing

To be able to determine who is fingerprinting and who is not we first need to be able to work with the code we gathered. The first step in this process is to unminify and possibly deobfuscate the code as most of the code we will gather will be minified and some will be obfuscated to a certain degree. We can do this by using a JavaScript deobfuscator like js-beautify[12]. This unminifies the code and deobfuscates the code, if possible, for us so we can more easily work with it.

The code that we have now is ready for manual analysis and we can determine who is fingerprinting with it. However, there is still a problem with the code if we want to automate this analysis process. This is a result of calls being split up in multiple parts, this happens when for example 'navigator.plugins' is stored to the variable 'n', and later in the code the call 'n.length' is made. During manual analysis it is not too hard to see that actually 'navigator.plugins.length' has been called, but when you automate this process it becomes less clear. To make the code ready for automation we make use of an Abstract Syntax Tree (AST). We construct the AST of the code and from this we can expand the member expressions, like done by van Zalingen et al. [vZH18]. This way the variables can be derived and we can see what calls are made in an automated way.

3.3.2 Analysing

To analyse who is fingerprinting and who is not we make use of the methods which we know are used by fingerprinters. We can look through the code to see which of these methods are being called. However, most of these methods are not exclusively used by fingerprinters. So we need a way to determine if some code using some of these methods is fingerprinting or simply using them for a different purpose. We do this by using a score mechanism which will give each script a score based on the calls the scripts makes. A higher score is assigned for a method which is more exclusively used for fingerprinting. This way we can count the score for all the methods that are being used by the code and can flag them as a fingerprinter if their score surpasses a certain threshold. These scores and the threshold we can base on what we see from our manual analysis.

[12]https://github.com/beautify-web/js-beautify

3.3.3 Separating the commercial fingerprinters from the non-commercial fingerprinters

To separate the commercial fingerprinters from the non-commercial fingerprinters we need to manually analyse the links and its corresponding code to see what company is behind the code. Then we need to see if this company offers its services online.

If the company offers their services online and that is the way the company is trying to make a profit, like AddThis[13], then we say they are commercial fingerprinters. If the services are offered online but they are not used to earn a profit, like fingerprint2.js[14], we say they are non-commercial fingerprinters.

[13]https://www.addthis.com/

[14]https://github.com/Valve/fingerprintjs2

Chapter 4

Manual analysis

As mentioned previously to build a tool that can find fingerprinters we need to understand how we can find fingerprinters. We do this by manually analysing potential fingerprinters and seeing why we decide who is fingerprinting and not. This gives us the methods we based our conclusions on about who is fingerprinting and we can use these methods to make an automated tool that can do the analysis for us. But before we can get to that point we need to know what methods are being used at the moment by fingerprinters and which of these methods separate them from non-fingerprinters.

4.1 Finding potential fingerprinters

In section 3.2 we named some different methods of finding potential fingerprinters which we use here to actually find potential fingerprinters.

The first way we find potential fingerprinters is by finding their advertisements on Google. We do this by googling the terms 'fraud prevention service online', 'online fraud prevention service', online fraud detection service', 'fraud detection service online', and 'personalized ad service online'. These terms give us direct hits on Google for companies advertising their services, but also other sites that sum up companies which do personalization, fraud detection and/or prevention, and authentication. We can then check the sites of the companies to see if they say anything that indicates that they are fingerprinting, these can be sentences like 'personalized in real time using browsing behavior, location, referring URLs, and more'[1] or 'Our

[1]https://uk.marketo.com/software/web-personalization/

system leverages device fingerprinting and velocity monitoring'[2].

In addition to that we can find potential fingerprinters in papers like the ones from Fouad et al. [FBLS18], Nikiforakis et al. [NKJ$^+$13], and Torres et al. [TJM15]. This will give us some companies that are known to have fingerprinted in the past. Because it is likely that these companies still have the same business model it is also likely that they still fingerprint in one way or another. This gives us some more potential fingerprinters.

Another way to find potential fingerprinters come from the potential fingerprinters that we have already found. Because we have to collect up-to-date JavaScript of the potential fingerprinters we need to scrape certain sites that make use of their services. In addition to the JavaScript of the potential fingerprinters we are looking for this also gives us all other executed JavaScript from these sites. In this extra information might be some other fingerprinters. Because we do not need to get all the fingerprinters out of this extra set we do not manually check each script for fingerprinting. However some of these scripts have suspicious names containing terms as 'tracking' or 'ad', we can see what company these scripts belong to and add them to the potential fingerprinters as there is a good chance that if we analyse these scripts we find some fingerprinting activities.

Using mainly the first method of simply googling for them, with a few added by the second method of known fingerprinters, a list of 25 potential fingerprinters was made to base the score system of our tool on, this list can be found in figure A.1. We later base our design in Chapter 5 on this list.

4.2 Manually checking if they actually fingerprint

Now we have a list of potential fingerprinters, but to be able to determine what methods the fingerprinters actually use and what we can recognise them by we need to analyse which of these potential fingerprinters actually fingerprint. To do this we first need the code that is deployed by these companies. To get this code we google 'sites that use *x*', where *x* is the name of the company we are looking for. Most of the time this gives us a list of sites that are apparently using the service of this company. However not all sites on this list actually use this service, so we scrape the first 10 sites named to have a better chance at finding the code of this company. In the case that we have not found the code yet in the first 10 we scrape the next 10 sites.

[2]https://www.riskified.com/solution/account-protection/

Now that we have the code deployed by the companies we can analyse it to see if they are fingerprinting or not. Because the JavaScript we get is often minified and sometimes obfuscated we "beautify" it before working with it. For this we use JavaScript beautifier[3], for which the code is also available online[4]. This way the code should be more readable and we can actually analyse it.

Using JavaScript beautifier we can transform the code from Figure 4.1, which is the code we collect, into a more readable piece of code that looks like the code in Figure 4.2

```
var bmo_lme=function(e){var n={};function t(o){if(n[o])return n[o].exports;var r=n[o]={i:o,l:!1,exports:{}};return e[o].call(r.exports,r,r.exports,t),r.l=!0,r.exports}return t.m=e,t.c=n,t.d=function(e,n,o){t.o(e,n)||Object.defineProperty(e,n,{configurable:!1,enumerable:!0,get:o})}
```

Figure 4.1: Minified code of a script found on bmo.com

```
var bmo_lme = function(e) {
    var n = {};

    function t(o) {
        if (n[o]) return n[o].exports;
        var r = n[o] = {
            i: o,
            l: !1,
            exports: {}
        };
        return e[o].call(r.exports, r, r.exports, t), r.l = !0,
            r.exports
    }
    return t.m = e, t.c = n, t.d = function(e, n, o) {
        t.o(e, n) || Object.defineProperty(e, n, {
            configurable: !1,
            enumerable: !0,
            get: o
        })
    }
```

Figure 4.2: Code from Figure 4.1 after beautifying

JavaScript beautifier can however not transform every piece of code into decently readable code as we can see from the code in Figure 4.3 which also

[3]https://beautifier.io/
[4]https://github.com/beautify-web/js-beautify

went through the JavaScript beautifier.

```
RISKX[_0xf649('0xb7')] = function(_0x2d32de) {
var _0x27b67a = {},
    _0x4d4d69 = null,
    _0x43ec53 = null,
    _0x56e033 = null;
try {
    var _0x49bc59 = document[_0xf649('0x40')](_0xf649('0xd9')),
        _0x20b93a = _0x49bc59['getContext'](_0xf649('0xda')) || _0x49bc59['getContext'](_0xf649('0xdb')),
        _0x58c33c = _0x20b93a[_0xf649('0xdc')](_0xf649('0xdd'));
    _0x4d4d69 = _0x20b93a[_0xf649('0xde')](_0x58c33c[_0xf649('0xdf')]), _0x43ec53 = _0x20b93a[_0xf649('0xde')](_0x58c33c[_0xf649('0xe0')]);
} catch (_0x2c983b) {
    _0x56e033 = _0x2c983b['message'];
}
_0x4d4d69 && (_0x27b67a[_0xf649('0xe1')] = _0x4d4d69), _0x43ec53 && (_0x27b67a[_0xf649('0xe2')] = _0x43ec53), _0x56e033 && (_0x27b67a[_0xf649('0xc3')] = _0x56e033), _0x2d32de[_0xf649('0xda')] = _0x27b67a;
}
```

Figure 4.3: Obfuscated code of Riskified after beautifying

This does, however not mean that we are not able to analyse any of the code. Some calls cannot be obfuscated, which means we might still encounter parts of the code like Figure 4.4, from which we can determine that the code is using the battery API, which can be used to fingerprint a user [OEN17].

```
RISKX['getBatteryJson'] = function(_0x36246e) {
    RISKX[_0xf649('0xa4')][_0xf649('0xc3')] ? _0x36246e[_0xf649('0xc4')] = RISKX['batteryData'][_0xf649('0xc3')] : RISKX['batteryData']['level'] && (_0x36246e['battery_charging'] = RISKX[_0xf649('0xa4')][_0xf649('0xc5')], _0x36246e[_0xf649('0xc6')] = RISKX['batteryData']['level'], RISKX['batteryData'][_0xf649('0xc5')] ? RISKX[_0xf649('0xa4')][_0xf649('0xc7')] && (_0x36246e['battery_charging_time'] = RISKX['batteryData'][_0xf649('0xc7')]) : RISKX[_0xf649('0xa4')]['dischargingTime'] && (_0x36246e[_0xf649('0xc8')] = RISKX['batteryData']['dischargingTime']), _0xf649('0xc9') == _0x36246e[_0xf649('0xca')] && (_0x36246e[_0xf649('0xca')] = -0x1), 'Infinity' == _0x36246e[_0xf649('0xc8')] && (_0x36246e[_0xf649('0xc8')] = -0x1));
}
```

Figure 4.4: Obfuscated code from Riskified using the battery API

Looking at newly found JavaScript code from a previously known fingerprinter, AddThis, we can find some interesting calls which might tell us something about if they are still fingerprinting or not. We might see some more innocent calls also often made by non-fingerprinters like screen resolution, which we can see in Figure 4.5 and user agent.

```
function(e, t, n) {
    "use strict";
    var a = n(5),
        i = n(422);
    e.exports = function(e, t, n, o, r) {
        var s = t || 550,
            d = n || 450,
            u = screen.width,
            c = screen.height,
            l = Math.round(u / 2 - s / 2),
            f = 0;
        c > d && (f = Math.round(c / 2 - d / 2));
        var p = window.open(e, a("msi") ? "" : o || "
            addthis_share", "left=" + l + ",top=" + f + ",width="
             + s + ",height=" + d + ",personalbar=no,toolbar=no,
            scrollbars=yes,location=yes,resizable=yes");
        return i.push(p), !!r && p
    }
}
```

Figure 4.5: Code from AddThis using the screen resolution

However we might also find some more interesting calls which a non-fingerprinting script might not make as often, like plugin enumeration, which we can see in Figure 4.6

```
r = navigator.plugins;
try {
    if (e = r.length, e > 0)
        for (var s = 0; s < Math.min(10, e); s++) s < 5 ? n += r
            [s].name + r[s].description : o += r[s].name + r[s].
            description
} catch (e) {}
```

Figure 4.6: Code from AddThis using the plugin enumeration

Just because some of these calls are more innocent, like screen resolution and user agent does not mean we cannot identify fingerprinters by looking at these functions. The way these calls are bundled with other calls can tell us something about if the script is fingerprinting or not. In figure 4.7 we can see the two named innocent calls being bundled with some other calls, showing that the innocent calls are part of a fingerprint here.

```
function(e, t, n) {
    var a = n(365),
        i = window;
    e.exports = function() {
        var e, t = a(navigator.userAgent, 16),
            n = (new Date).getTimezoneOffset() + "" + navigator.
                javaEnabled() + (navigator.userLanguage ||
                navigator.language),
            o = i.screen.colorDepth + "" + i.screen.width + i.
                screen.height + i.screen.availWidth + i.screen.
                availHeight,
            r = navigator.plugins;
        try {
            if (e = r.length, e > 0)
                for (var s = 0; s < Math.min(10, e); s++) s < 5
                    ? n += r[s].name + r[s].description : o += r[
                    s].name + r[s].description
        } catch (e) {}
        return t.substr(0, 2) + a(n, 16).substr(0, 3) + a(o, 16)
            .substr(0, 3)
    }
}
```

Figure 4.7: Code from AddThis using screen resolution and user agent as part of a fingerprint

Now that we have seen some of the calls made by a script we need to determine if they are fingerprinting or not. To do this we look at what calls are being made and what is being done with the information that these calls give. If there are calls made which are rarely used by non-fingerprinters, like battery calls and oscillator calls, this raises the suspicion of these scripts. However we still need to see what is done with this information because these calls also have purposes other than fingerprinting. If we see that the script stores this information in a part of an ID or in a variable, which then only gets used to send the information somewhere we can say this script is fingerprinting.

We might also encounter scripts which do not call any of these functions rarely used by fingerprinters. To see if these scripts are fingerprinting we thus need a different approach. For these scripts we look at the more normal calls, what they are being used for and how often they are called. If we see a function like 'screen.width' being called this might be because the script need the width to display something. It might also be the case that they want the width to use it as part of a fingerprint, thus we need to look if the width is being used for a non-fingerprinting purpose like displaying something. If this is the case the script is not using this function for fingerprinting. It can also be the case that they first use the width to display something, but then later on in the script use the 'screen.width' call five more times. This means we need to see what these five extra calls are being used for, which is quite possibly fingerprinting.

Using these methods we have found multiple fingerprinters, of which randomly picked 5 to investigate more closely to see what methods they use. The results of this can be found in Figure 4.8

4.3 Results

Using this method to analyse the potential fingerprinters we can determine that there are fingerprinters among them, and we can see which methods they are using. In Figure 4.8 we have put 5 of these fingerprinters, which were chosen randomly from the fingerprinters we found in section 4.2, along with an overview of the methods they are using. We will later use this overview in Chapter 5 for our tool.

Attribute	AddThis	Fervor	Riskified	iovation	Simility
Plugin Enumeration	✓	✓		✓	✓
User-Agent	✓	✓		✓	✓
Screen Resolution	✓	✓		✓	✓
Timezone	✓		✓		✓
Browser Language	✓	✓		✓	✓
DOM Storage	✓			✓	✓
Java Enabled	✓				
DNT User Choice	✓				✓
Cookies Enabled	✓	✓	✓	✓	✓
JS detect: Flash Enabled				✓	
Date & Time	✓	✓	✓	✓	✓
System/User Language	✓	✓		✓	✓
OpenDatabase				✓	✓
Canvas Fingerprint	✓				✓
IndexedDB			✓		
Device Identifiers					✓
IP address			✓		
Battery			✓		
WebGLRenderingContext					✓
GeoLocation	✓				
LocalStorage	✓		✓	✓	

Figure 4.8: Overview of methods used by these fingerprinters

Based on these findings we created the following scoring system, where a method gets a score based on how likely it is that the script is fingerprinting if we find that method. This takes into account how often the method is used by fingerprinters as well as non-fingerprinters. The threshold was then set at 10 by looking at what scores different combinations would result in and seeing if they should be seen as fingerprinting or not.

If the script uses device identifiers or bundles information together in a single function like in figure 4.7 we give the script a score of 10.

If the script uses the battery API, a plugin enumeration, or uses the canvas just to extract a value out of it, we give this script a score of 9 for each.

If the script requests the screen width, height, color depth and pixel depth, we give this script a score of 7.

If the script requests all kind of languages(browser, user, system), or the doNotTrack user choice, we give this script a score of 5 for each.

If the scripts uses the local storage, java enabled, cookie enabled functions, or the screen width, height, and color depth, we give this script a score of 2 for each.

If a script has a total score of 10 or more we say the script is fingerprinting. We put this at 10 or more because this way if a script creates a fingerprint it immediately has this score, however if a script collects some information that is often used for fingerprinting it needs to collect multiple before it actually reaches the score for fingerprinting.

Chapter 5

Design

Doing this collection and analysis by hand, as we did in Chapter 4, for a large set of sites is not feasible, so we design a tool for this which collects the data and then analyses it.

We want to analyse JavaScript files, so to do this we first need to collect some JavaScript files. We do not want to analyse the same files multiple times, as this would only cost more effort without bringing any benefit. We also want the code from these files to be unminified and deobfuscated if possible, so that we can expand the member expressions and derive the variables. This way we can see the full calls that have been split up in the code. This way we can actually analyse the full calls made by the script.

The first thing to do is collect some JavaScript files. For this we use OpenWPM, which will scrape a set of websites for us, and collect links to all the JavaScript that gets executed during its visit. It will also give us a set of deduplicated links, which helps with not analysing the same files multiple times.

Our tool will be available online at GitHub[1].

5.1 Preparing the files

Now we want to collect the actual JavaScript files that we have the links of so that we can collect the code from it and analyse this code. To collect the code via the links we obtain with OpenWPM we use the python wget

[1]https://github.com/Bart-v-V/ThesisTool

package[2]. This will give us the JavaScript file that the link we give it points to.

Even though we have a set of deduplicated links it is still possible that different links provide us with the same code. To make sure we do not analyse the same code multiple times we create a hash of the code we collect. We then check if we have seen the hash before, in the case that we have seen the hash before we know that we have seen the code before and do not analyse it. If we have not seen the hash before we analyse the code and save the hash to compare future hashes with.

Before analysing the code we want to unminify and deobfuscate the code so that we can expand member expressions and derive variables from the code. This way we can actually see and analyse the full calls made in the code. For this purpose we use the python package JS Beautifier[3]. This takes the code and unminifies it, and also deobfuscates it when possible.

Now we can expand member expressions and derive variables from the code so that we can analyse what calls are being made. To do this we create an abstract syntax tree (AST) of the code, from this AST we can then expand the member expressions and derive the variables. For this we use the code made by van Zalingen et al. [vZH18], which is available online[4], we updated and edited this code to fit our purpose.

5.2 Detection mechanics

We want to be able to detect which scripts are fingerprinting. To do this we want a scoring mechanism which based on the score can tell us if a script is fingerprinting. For this we need some kind of mechanism to detect activities that suggest fingerprinting. So we need a list of activities that suggest fingerprinting, similar as to what we did in section 4.3, however now with a subset of the signs used in section 4.3 so that we can automate it.

For our list of activities that suggest fingerprinting we use some of the signs of a fingerprinter laid out by Laperdrix in his PHD thesis [Lap17]. We use the following signs:

1. Accessing specific functions
2. Hashing values

[2]https://pypi.org/project/wget/
[3]https://github.com/beautify-web/js-beautify
[4]https://github.com/Timvanz/static-javascript-fingerprint-classification

3. Creating an ID
4. sending information to a remote address

Not all signs laid out by Laperdrix were used because some of the signs are a lot harder to detect in an automated way, or it is hard to determine if the sign was actually used for fingerprinting of for a different purpose in an automated way. These four signs were chosen because we can detect them in an automated way and from our manual analysis we have seen that we can identify a fingerprinter with them.

For the first, 'accessing specific functions', we want to have some function which if used, are most likely used for fingerprinting. Based on the findings of previous papers and what we saw during our manual analysis, we choose to interpret the following function calls as signs of fingerprinting:

1. battery functions, consisting of 'navigator.getBattery()', 'battery.charging', 'battery.level', 'battery.chargingTime', and 'battery.dischargingTime'.
2. plugin functions, consisting of 'navigator.plugins', 'plugins.name', 'plugins.length', and 'plugins.description'
3. mimeType functions, consisting of 'navigator.mimeTypes', 'mimeType.enabledPlugin', 'mimeType.description', and 'mimeType.type'
4. navigator.doNotTrack and navigator.msDoNotTrack
5. screen resolution functions, consisting of 'screen.width', 'screen.height', and 'screen.colorDepth'
6. navigator.platform
7. navigator.cookieEnabled
8. navigator.javaEnabled
9. date.getTimeZoneOffset
10. window.localStorage
11. window.sessionStorage

For the other three signs we want to have specific keywords which suggests that the code has any of these three signs. For the second sign, 'hashing values', we decided on the keyword "hash", based on what we saw during our manual analysis. For the third sign, 'creating an ID', we decided on the keywords "identifier" and "userID", based on what we saw during our manual analysis. For the fourth sign, 'sending information to a remote address', we decided on the keywords "beacon" and "GetDataURL", based on what we saw during our manual analysis. Using "beacon" here means that we also catch the user of the function call 'sendBeacon()', which also gets used to send information to a remote address. Then we have some keywords which are often used by fingerprinters, which are added based on findings of previous papers. We have "ad.js" which gets used to test if an ad blocker is installed, which is mostly used by fingerprinters. "WebGL_debug_renderred_info" and "oscillator" are chosen because they are almost exclusively used by fingerprinters, while they do have non-fingerprinting functionality, it is rarely needed and thus rarely used by non-fingerprinters.

Using obfuscation it is quite easy to avoid getting detected by our tool, as we use the python package JS Beautifier to deobfuscate, so to avoid detection one could simply obfuscate in a way that JS Beautifier can not deobfuscate. We could however deobfuscate more kinds of obfuscations, as shown by Gabry Vlot [Vlo18]. However we left that out of the scope of this thesis.

5.2.1 Scoring mechanism

To determine if a script is fingerprinting, using these function calls and keywords, we need a scoring mechanism. So we need a score for each function call and keyword and a threshold for saying that a script is fingerprinting using these calls and keywords. The scores we give the function calls and keywords are based on how often a call of keyword is used for fingerprinting in comparison to how often it is used for non-fingerprinting purposes, and how likely they thus are to indicate a fingerprinter in comparison with the other calls and keywords. This gave us the scoring table in figure 5.1. Most of these function calls / keywords are related to the attributes in figure 4.8, and the choice for the other is explained earlier in section 5.2. These scores are however not directly related to the scores used in section 4.3, because of the addition of the new function calls and keywords, and the absence of some other used in section 4.3, we chose to determine the scores in table 5.1 from the ground up. We do however see that when a method has a higher score compared to another method in section 4.3, it also has a higher score in table5.1 compared to that other method.

For numbers 1 through 4 we count the score given to this function once one of the considered calls has been made. This has been done because these calls are often used for fingerprinting without using all of the calls, however

	Function call / Keyword	Score
1.	battery	12
2.	plugins	7
3.	mimeTypes	6
4.	navigator.doNotTrack and navigator.msDoNotTrack	8
5.	screen resolution	4
6.	navigator.platform	6
7.	navigator.cookieEnabled	4
8.	navigator.javaEnabled	4
9.	date.timeZoneOffset	1
10.	window.localStorage	2
11.	window.sessionStorage	2
12.	"hash"	6
13.	"identifier"	4
14.	"userID"	6
15.	"beacon"	6
16.	"GetDataURL"	4
17.	"ad.js"	8
18.	"WebGL_debug_renderred_info"	9
19.	"oscillator"	12

Figure 5.1: Scoring table

these calls still have non-fingerprinting purposes, so we chose this way to not automatically flag a script as fingerprinting which uses multiple of these calls while also not flagging a script as non-fingerprinter because they only use one of the calls. For number 5 we only count the score if all the calls are made, this has been done because 'screen.width' and 'screen.height' are often used for non-fingerprinting purposes, however 'screen.colorDepth' is used a lot less for these purposes, so by only counting the score if all the calls are made we avoid counting the score for the calls for non-fingerprinting purposes, while still counting the score if it is more likely to be for fingerprinting. For the others we count the score if they occur in the code.

We constructed the scoring table in figure 5.1 using the following steps:

1. We started with the first entry and gave it a score of 2, this way we could go higher or lower for the next entries.
2. For the next entry we compared it to the scores of existing entries to see which score it should get, based on how likely the entry is to be used by a fingerprinter compared to how likely it is to be used by a non-fingerprinter.

3. Up the score of other entries if the score of an entry should be between two other entries of which the score have no integer in between.

Now we are able to obtain a total score for a script by adding all the scores of that script together. We then want to determine if the script is capable of fingerprinting using this score. For this we need a threshold. We ran our tool on 20 manually analysed scripts, of which 10 fingerprinting and 10 non-fingerprinting, for the non-fingerprinting scripts it gave total scores in the range of 0-16. While for the fingerprinting scripts it gave results in the range of 19-71. Based on these results we set out threshold on 17.

Scripts that get a score greater or equal to the threshold are flagged as *sufficient information*, meaning that this script collects enough data to be able to fingerprint, thus meaning that this script is possibly fingerprinting. Scripts under the threshold are flagged *insufficient information* meaning that this script does not collect enough information to fingerprint. If we encounter a script we have already analysed it gets flagged as *duplicate*. If we fail to collect a script, for whatever reason, we flag it as *fails*.

5.3 Automation

We want to automate this process because doing it manually for a large number of links is not feasible. For this we need a part that collects the code, a part that hashes the code and checks if we have seen it before, a part that creates the AST, expands the member expressions and derives the variables, and a part that counts the scores and flags the scripts accordingly.

These first two, collecting the code and hashing, we can combine together in a single python script. To this we can also add the score counting for the keywords. After that we need a different script which creates the AST, expands the member expressions and derives the variables, for which we use a JavaScript script. Last we need a script which counts the score for the function calls, add this to the previous calculated score for the keywords and determines if the script is fingerprinting. For this we use another python script.

Then finally to be able to do this for a large number of links we create a bash script which runs these three scripts for each link.

Chapter 6

Case study

To see how well our tool from Chapter 5 actually works, we use it in a case study. This will show the things our tool does right and the shortcomings of our tool.

6.1 Methods

To run our tool we need to have data to put into it. We need to provide our tool with a set of links to JavaScript scripts to do its job. To collect these links we used OpenWPM [EN16]. OpenWPM needs a set of sites to go to, we used the Tranco list [PvGT^{+}19] to get this set of sites.

We used the Tranco list created on 28 October 2019, which is available online[1]. From this list we took a total of 10.000 sites that we used. To get these 10.000 we took the first 5.000 sites and used a python program using the python random() function to get 5.000 numbers between 5.000 and 20.000. Then we grabbed the 5.000 sites from the Tranco list that corresponded with the random numbers generated and added these to the top 5.000 sites.

These 10.000 sites were then used as input for OpenWPM, which was setup go to each of these sites and browse to one extra page for each site. It was also setup to collect the links to all the JavaScript that was executed and to not run in headless mode. We ran OpenWPM in its stateless mode and used the standard Firefox browser with Flash enabled.

[1]https://tranco-list.eu/list/KWVW

OpenWPM gives us a set of links which is deduplicated. But because we had to split up the data collection with OpenWPM into multiple parts, due to storage space limitation, it is possible to have a link in multiple sets. This means that when we combine these sets to get the full set over the 10.000 there might be duplicate links, so we deduplicated the full set before moving on.

This gave us a set of 85.041 distinct links to JavaScript files, on which we ran our tool.

We then took a part of the list of links flagged as *sufficient information* and manually looked at them to see if we could find commercial fingerprinters. This was done by first filtering out some known non-commercial fingerprinters and known commercial fingerprinters, including those in Figure A.1, that got flagged by our tool. Then we looked at the link and code to see if we could find the company behind it, if we found a company we looked at the code to confirm it was actually fingerprinting. If the script was actually fingerprinting we looked to see if the company was offering its services online, if it does offer its services online we say it is a commercial fingerprinter and add it to our list of previously unknown commercial fingerprinters.

6.2 Results

After our tool analysed the links provided it flagged these scripts in the following way:

- flagged 9.835 scripts as sufficient information
- flagged 41.262 scripts as insufficient information
- found 29.098 duplicate scripts
- failed to collect 2.391 scripts
- 2.455 scripts that we lost, which was discovered after further inspection

Using a part of the set of links from the scripts flagged as sufficient information and our method to find commercial fingerprinters we found 13 commercial fingerprinters which were previously unknown to us, namely:

1. Getintent[2]

[2]https://getintent.com/

2. PubMatic[3]
3. Monetate[4]
4. Adform[5]
5. Akamai[6]
6. Ptengine[7]
7. Meetrics[8]
8. ONEcount[9]
9. Duo[10]
10. PerimeterX[11]
11. Sophi[12]
12. GumGum[13]
13. TalkingData[14]

6.3 Analysis of results

Our tool has given us the results shown above, now we will look at what these results actually mean and what they do not mean.

The result *sufficient information* means that the script we analysed collects enough information to be able to fingerprint. This does not necessarily mean that this script actually fingerprints. Some of these scripts might be tracking in a different way instead of actually fingerprinting. Another options is that these scripts are run to collect certain statistics, so they might collect some of the information a fingerprinter would, but never actually

[3]https://pubmatic.com/
[4]https://monetate.com/
[5]https://site.adform.com/
[6]https://www.akamai.com/
[7]https://www.ptengine.com/
[8]https://www.meetrics.com/en/
[9]https://www.one-count.com/
[10]https://duo.com/
[11]https://www.perimeterx.com/
[12]https://sophi.io/#top
[13]https://gumgum.com/
[14]http://www.talkingdata.com/

create a fingerprint. Included in the list of scripts flagged as sufficient information are multiple previously known fingerprinters.

Insufficient information means that the analysed script does not collect enough of the information we looked at to create a fingerprint out of that. However, we have not looked at every possible piece of information that can be used to create a fingerprint. This means that even though our tool flagged it as insufficient information it might actually create a fingerprint using information that our tool does not look at. The script can also be too obfuscated or split up calls without dots. It is also possible that not all member expressions are expanded and thus we miss them. Included in the list of insufficient information are scripts from Riskified, which due to their obfuscation are not flagged as sufficient information, even though they collect enough information and actually are fingerprinting.

Duplicate means that the script produced a hash which was also produced by a script we previously analysed. This most likely means that it is also the same script as we previously analysed, but than with a different link. However there is a chance to get the same hash with a different script as the output of the hash function is a static size. We have 16^{32} different possible hash results and 85.041 scripts which are hashed. This means that there is a $85.041/16^{32}$, which is about 1 in $4 \cdot 10^{36}$, chance that there is a hash collision, which is negligible.

Fails means that the tool somehow failed to collect the code. This can be for a few different reasons, the first is that the link is in such a weird format that the python wget package can not understand it and will thus not be able to download the code. Another possibility is that the link we have gives us a 401 not authorized error, in which we are also not able to access the code. We can also get a timeout error if the download takes too long, which can be the case if the link can not be reached. The wget package also has a limit of the length of the link it can handle, which means that some of the link were simply too long for the wget package and it thus failed to collect the code.

Our tool reported a total of 2.391 scripts it failed to collect. However, if we add the total amount of scripts our tool reported on we see something weird, as it reported on a total of 82.586 script, while we put in a list of 85.041 scripts, which means we lost a total of 2.455 script somewhere. On further inspection it seems that all these scripts give decoding errors. This then results in our tool not being able to analyse them, and because of that our tool did not flag these scripts at all.

Finding 13 previously unknown commercial fingerprinters in a part of the list flagged as 'sufficient information' means that our tool actually flagged these fingerprinters correctly. However there are also still a lot of non-fingerprinters(e.g. analytical scripts and other trackers) flagged as sufficient information. Which means that our tool works, but more like tool that makes a preselection of scripts then a fingerprinter finder, as there is still quite some manual work to do to actually find the fingerprinters.

Next to the 13 previously unknown commercial fingerprinters we also found multiple questionable scripts which might be fingerprinting, but we are not totally sure they are actually fingerprinting. Because we are not totally sure they are not added to the list of previously unknown commercial fingerprinters.

Chapter 7

Conclusions

We have shown in Chapter 6 that it is possible to partially separate fingerprinters from non-fingerprinters using static JavaScript analysis. It is however harder to separate some fingerprint-like activities, like analytical scripts and different tracking, from actual fingerprinters this way. So it looks like it is not possible to separate those from the fingerprinters without errors using static JavaScript analysis. It does however work as a preselection. This can be useful to reduce the amount of work needed to find the actual fingerprinters. In section 6.3 we used our methods to identify 13 previously unknown commercial fingerprinters, which we manually verified were actually commercial fingerprinters, this shows that our method works. So this method works, but it is probably not too useful as we speculate that there are better alternatives like the way FP-Detective [AJN$^+$13] did, where they can also find unknown fingerprinters but they look more towards a single method being used. Looking at a single method used might not find you the same fingerprinters and it might also not find you the same amount of fingerprinters. However it does result in a lot less noise in the list of *sufficient information* scripts. And thus requires less manual work after using the tool. Which makes it easier to find the actual fingerprinters in the list of *sufficient information*, and we speculate that it would also require less manual work to find the same fingerprinters we found by running a tool focused on a single method multiple times, looking for a different method each time. As these are speculations, this is something that would have to be looked into to see how well it actually works.

Bibliography

[AJN$^+$13] Gunes Acar, Marc Juárez, Nick Nikiforakis, Claudia Díaz, Seda F. Gürses, Frank Piessens, and Bart Preneel. Fpdetective: dusting the web for fingerprinters. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1129–1140. ACM, 2013.

[AvO16] Furkan Alaca and Paul C. van Oorschot. Device fingerprinting for augmenting web authentication: classification and analysis of methods. In Stephen Schwab, William K. Robertson, and Davide Balzarotti, editors, *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, pages 289–301. ACM, 2016.

[EN16] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1388–1401. ACM, 2016.

[FBLS18] Imane Fouad, Nataliia Bielova, Arnaud Legout, and Natasa Sarafijanovic-Djukic. Tracking the pixels: Detecting web trackers via analyzing invisible pixels. *CoRR*, abs/1812.01514, 2018.

[JKV19] Hugo Jonker, Benjamin Krumnow, and Gabry Vlot. Fingerprint surface-based detection of web bot detectors. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-*

27, 2019, Proceedings, Part II, volume 11736 of *Lecture Notes in Computer Science*, pages 586–605. Springer, 2019.

[Lap17] Pierre Laperdrix. *Browser Fingerprinting: Exploring Device Diversity to Augment Authentication and Build Client-Side Countermeasures. (Empreinte digitale d'appareil: exploration de la diversité des terminaux modernes pour renforcer l'authentification en ligne et construire descontremesures côté client).* PhD thesis, INSA Rennes, France, 2017.

[Lib15] Timothy Libert. Exposing the hidden web: An analysis of third-party HTTP requests on 1 million websites. *CoRR*, abs/1511.00619, 2015.

[MM12] Jonathan R. Mayer and John C. Mitchell. Third-party web tracking: Policy and technology. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 413–427. IEEE Computer Society, 2012.

[NJL15] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. Privaricator: Deceiving fingerprinters with little white lies. In Aldo Gangemi, Stefano Leonardi, and Alessandro Panconesi, editors, *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 820–830. ACM, 2015.

[NKJ$^+$13] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 541–555. IEEE Computer Society, 2013.

[OEN17] Lukasz Olejnik, Steven Englehardt, and Arvind Narayanan. Battery status not included: Assessing privacy in web standards. In José M. del Álamo, Seda F. Gürses, and Anupam Datta, editors, *Proceedings of the 3rd International Workshop on Privacy Engineering co-located with 38th IEEE Symposium on Security and Privacy, IWPE@SP 2017, San Jose, CA, USA, May 25, 2017.*, volume 1873 of *CEUR Workshop Proceedings*, pages 17–24. CEUR-WS.org, 2017.

[PvGT$^+$19] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *26th Annual Network and Distributed System Security*

Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019. The Internet Society, 2019.

[Roe14] Franziska Roesner. *Security and Privacy from Untrusted Applications in Modern and Emerging Client Platforms*. PhD thesis, University of Washington, 2014.

[TJM15] Christof Ferreira Torres, Hugo L. Jonker, and Sjouke Mauw. Fp-block: Usable web privacy by controlling browser fingerprinting. In Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl, editors, *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*, volume 9327 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2015.

[Vlo18] Gabry Vlot. Automated data extraction; what you see might not be what you get. Master's thesis, Open University, the Netherlands, 2018.

[vZH18] Tim van Zalingen and Sjors Haanen. Detection of browser fingerprinting by static javascript code classication. University of Amsterdam, 2018. Research Project, https://rp.delaat.net/2017-2018/p82/report.pdf.

Appendix A

Potential fingerprinters

Here we listed the 25 potential fingerprinter which we found in section 4.1. The potential fingerprinters were mostly found by googling, with AddThis, iovation, and LexisNexis coming from previous papers. We use this list in section 4.2 to find actual fingerprinters.

AddThis https://www.addthis.com/	Kount https://www.kount.com/	Kaspersky https://usa.kaspersky.com/
RSA https://www.rsa.com/	Emailage https://emailage.com/	FRISS https://www.friss.com/
IPQualityScore https://www.ipqualityscore.com/	TransUnion https://www.transunion.com/	LexisNexis https://www.threatmetrix.com/
Simility https://simility.com/	Riskified https://www.riskified.com/	Signifyd https://www.signifyd.com/
Sift https://sift.com/	Forter https://www.forter.com/	FraudLabs Pro https://www.fraudlabspro.com/
DupZapper https://dupzapper.com/	Bolt https://www.bolt.com/	PPC Protect https://ppcprotect.com/
iovation https://www.iovation.com/	Ensighten https://www.ensighten.com/	Tealium https://tealium.com/
AdRoll https://www.adroll.com/	Fervor https://www.createfervor.com/	Adscore https://www.adscore.com/
	Matomo https://matomo.org	

Figure A.1: Potential fingerprinters found during the manual analysis of section 4.1